줄리아 언어

The Julia Project

February 16, 2021

Contents

Part I

매뉴얼

Chapter 1

줄리아 1.5 문서

환영합니다. 줄리아 1.5 문서입니다.

> 번역 안내
>
> 한글 문서 번역은 깃헙 https://github.com/juliakorea/translate-doc 에서 누구나 참여하실 수 있습니다. 많은 참여 부탁드립니다.
>
> 구글 자동 번역
>
> 중국어 번역 자료를 한국어로 자동 번역해서 유용하게 볼 수 있습니다.
>
> - Julia 대만(juliatw) 문서 https://docs.juliatw.org/latest/
> - Julia 중국(juliacn) 문서 http://docs.juliacn.com/latest/
>
> Note
>
> 이 문서를 PDF 형태로 보실 수 있습니다: julia-1.5.1.pdf.

소개 글

과학 분야 컴퓨팅은 빠른 성능을 요구함에도, 정작 대부분의 연구자들은 속도가 느린 동적인 언어로 일을 처리한다. 동적 언어를 즐겨쓰는 여러 이유로 보아 이러한 추세는 쉽게 사그러들지는 않아 보인다. 다행히 언어 디자인과 컴파일러 기법의 발달로 성능 문제가 해결되면서 동적 언어의 성능 하락 문제를 극복하고 프로토타이핑과 계산 집중형 애플리케이션의 구축을 하나의 환경에서 발휘할 수 있게 되었다. 줄리아 (Julia)는 이런 장점을 최대화한 언어이다. 줄리아는 (1) 과학과 수학 분야의 컴퓨팅에 적합한 동적 언어이면서 (2) 정적 타입 언어에 견줄만한 성능을 지닌다.

줄리아 컴파일러는 파이썬, R과 같은 언어의 해석 방식과 다르다. 줄리아가 뽑아내는 성능이 아마도 처음에는 의아할 것이다. 그럼에도 작성한 코드가 느리다면 성능 팁을 읽어보길 권한다. 줄리아가 어떤 식으로 작동하는지 이해한 뒤라면, C에 근접하는 성능의 코드를 짜는 건 쉽다.

줄리아는 타입 추론과 LLVM으로 구현한 JIT 컴파일을 사용해 선택적 타입, 멀티플 디스패치, 좋은 성능을 이뤄내고 있다. 그리고 명령형, 함수형, 객체 지향 프로그래밍의 특징을 포괄하는 다양한 패러다임을 추구한다. 줄리아는 고급 단계의 수치 계산에 있어 R, 매트랩, 파이썬처럼 간편하고 표현력이 우수하다. 뿐만 아니라 일반적인 형태의 프로그래밍 또한 가능하다. 이를 위해 줄리아는 수학 프로그래밍 언어를 근간으로 해서 구축하였고 리스프, 펄, 파이썬, 루아, 루비와 같은 인기있는 동적 언어의 기능을 취합하고 있다.

기존에 있는 동적 언어와 비교해 보는 줄리아만의 독특한 점은:

- 핵심 언어는 최소한으로 꾸린다; 정수를 다루는 프리미티브 연산자(+ - * 같은)를 포함하는, 줄리아 Base와 표준 라이브러리는 줄리아 코드로 짜여져 있다.
- 객체를 구성하고 서술하는 타입(types)을 풍부하게 지원하며, 타입 선언을 만들 때도 선택적으로 사용된다.
- 인자 타입을 조합함으로서 함수의 작동 행위를 정의하는 멀티플 디스패치(multiple dispatch)
- 인자 타입에 따라 효율적이고 특화된 코드를 자동으로 생성
- C와 같은 정적으로 컴파일되는 언어에 근접하는 훌륭한 성능

동적 언어에 대해 "타입이 없다"는 식으로 말하곤 하는데 사실은 그렇지 않다: 기본 타입이거나 별도 정의를 통해 모든 객체는 타입을 가진다. 그러나 대부분의 동적 언어는 타입 선언의 부족으로 컴파일러가 해당 값의 타입을 인지하지 못한다거나 타입에 대해 무엇인지 명시적으로 밝힐 수 없는 상태가 되곤 한다. 한편 정적 언어는 타입 정보를 – 대개 – 컴파일러용으로서 달기에, 타입은 오직 컴파일 시점에만 존재하여 실행시에는 이를 다루거나 표현할 수 없다. 줄리아에서 타입은 그 자체로 런타임 객체이며 컴파일러가 요하는 정보를 알려주는 데에도 쓰인다.

보통의 프로그래머라면 개의치 않을 타입과 멀티플 디스패치는 줄리아의 핵심 개념이다: 함수들은 서로 다른 인자 타입들을 조합함으로서 정의되고 가장 그 정의와 맞물리는 타입을 찾아 디스패치하여 실행된다. 이 모델은 수학 프로그래밍과 잘 맞는데, 전통적인 객체 지향 디스패치라면 첫번째 인자로 연산자를 "갖는" 것은 자연스럽지 않다. 연산자는 단지 특별히 표기한 함수일 뿐이다 – + 함수에 엮일 새로운 데이터 타입을 정의하려면, 해당하는 메서드 정의만 추가하면 된다. 기존 코드는 새로운 데이터 타입과 더불어 원할하게 작동한다.

런타임 타입 추론(타입 지정을 점진적으로 늘려가며)을 이유로, 또 이 프로젝트를 시작할 때 무엇보다도 성능을 강조하였기에 줄리아의 계산 효율은 다른 동적 언어들에 비해 우월하며 심지어 정적으로 컴파일하는 경쟁 언어들마저 능가한다. 거대 규모의 수치 해석 문제에 있어 속도는 매번 그리고 앞으로도, 아마 항상 결정적 요소일 것이다: 처리되는 데이터의 양이 지난 수십 년간 무어의 법칙을 따르고 있지 않은가.

사용하기 편하면서도 강력하고 효율적인 언어를 줄리아는 목표하고 있다. 다른 시스템과 견주어 줄리아를 씀으로 좋은 점은 다음과 같다:

- 자유롭게 사용 가능하며 오픈 소스이다 (MIT 라이센스)

- 사용자가 정의한 타입 또한 내장한 타입처럼 빠르고 간결하다
- 성능을 위해 코드를 벡터화할 필요가 없다; 벡터화하지 않은 코드도 빠르다
- 병렬과 분산 처리를 위해 고안되었다
- 가벼운 "그린" 쓰레딩 (코루틴)
- 거슬리지 않는 강력한 타입 시스템
- 숫자와 다른 타입을 위한 우아하고 확장 가능한 컨버젼 및 프로모션(타입 변환)
- 효율적인 유니코드 와 UTF-8 지원
- C 함수 직접 호출(별도의 래퍼나 특정한 API가 필요하지 않음)
- 다른 프로세스를 관리하는 쉘과 비슷한 강력한 기능
- 리스프 (Lisp)와 비슷한 매크로, 메타프로그래밍을 위한 장치들

Chapter 2

시작하기

줄리아의 설치는 어렵지 않다. 미리 만들어진 실행파일을 사용하거나, 소스로부터 직접 컴파일하는 두 방법이 있다. https://julialang.org/downloads/에서 알려주는 방법에 따라 Julia를 다운로드하고 설치하면 된다.

Julia 대화형 실행 환경(REPL)은 Julia를 가장 쉽게 익힐 수 있는 수단이다. 대화형 실행 환경은 단순히 Julia 실행파일을 더블 클릭하거나, 명령창에 `julia`를 입력해 실행할 수 있다.

```
$ julia


               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.3 (2020-11-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia> 1 + 2
3

julia> ans
3
```

대화형 실행 환경을 종료하기 위해서는 `CTRL-D`(컨트롤 키와 `d` 키를 함께 누른다) 를 누르거나 `exit()`를 입력한다. 대화형 실행 환경을 실행하면, 위와 같이 `julia` 배너가 보여지고, 커서창이 사용자의 입력을 기다리며 깜빡이고 있다. 사용자가 `1 + 2`와 같은 표현식을 입력한 뒤 엔터 버튼을 누르면 Julia는 표현식을 계산하고 결과를 보여준다. 만약 사용자가 입력한 표현식이 세미콜론(;)으로 끝난다면, 대화형 실행 환경은 결과를 화면에 보여주지 않는다. 대신 `ans` 라는 변수가 가장 마지막으로 계산된

표현식의 결과를 저장하고 있으므로 이를 활용할 수 있다. ans 변수는 대화형 실행 환경에서만 존재하며, 다른 방식으로 동작하는 Julia 코드 상에서는 나타나지 않는다.

file.jl라는 소스 파일에 저장되어 있는 코드를 실행하기 위해서는include("file.jl")을 입력하면 된다.

대화형 실행 환경을 이용하지 않고 파일에 저장되어 있는 코드를 실행하기 위해서는, 소스 파일 이름을 julia 명령어의 첫번째 매개 변수로 넣어서 실행한다.

명령어:

```
$ julia script.jl arg1 arg2...
```

예제와 같이 julia 실행 명령 뒤에 오는 매개변수들은 전역 상수 ARGS라고 불리우는 script.jl라는 프로그램의 명령줄 인자로 작동한다. 이 프로그램의 이름은 전역 상수 PROGRAM_FILE 에도 설정된다. 또한 ARGS는 이 뿐만이 아니라-e 옵션을 통해서 julia 스크립트를 실행할 때도 설정할 수 있음을 알 필요가 있다. 그러나 이 경우에는 PROGRAM_FILE 은 아무것도 설정되지 않은 채로 실행될 것이다. (아래의 julia 도움말을 보도록 하자.) 예를 들어, 단순히 스크립트에 주어진 명령줄 인자를 출력할 때는 다음과 같이 입력하면 된다.

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end' foo bar


foo
bar
```

아니면 저 코드를 스크립트 파일에 넣고 실행시켜도 가능하다.

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

-- 구분자는 명령어와 줄리아에 넘겨줄 인자를 구분하는데 사용한다.

```
$ julia --color=yes -O -- foo.jl arg1 arg2..
```

See also Scripting for more information on writing Julia scripts.

Julia는 -p 옵션이나 --machine-file 옵션을 이용하여 병렬 환경에서 실행시킬 수 있다. -p n 옵션은 n개의 worker 프로세스를 생성하지만, --machine-file file 옵션은 file의 각 행에 지정된 노드마다 worker를 생성한다. file 에 지정된 노드(machine)들은 ssh 로그인을 통해 패스워드가 필요없이 실행할 수 있어야 하며, Julia는 현재 호스트와 같은 경로에 설치가 되어 있어야 한다. file 에 작성되는 노드는 [count*][user@]host[:port] [bind_addr[:port]] 와 같은 형식으로 작성한다. user 는 현재 user id를 나타내고, port 는 기본 ssh port, count 는 각 노드당 생성하는 worker의 개수 (기본값 : 1) bin-to bind_addr[:port] 은 선택적인 옵션으로 다른 worker들이 현재의 worker로 연결하기 위해 필요한 특정 IP 주소와 포트를 지정한다.

만약 Julia가 실행할 때마다 실행되는 코드가 있다면, 그 코드를 `~/.julia/config/startup.jl` 에 넣으면 된다.

```
$ echo 'println("Greetings!  好! 안녕하세요?")' > ~/.julia/config/startup.jl
$ julia
Greetings!  好! 안녕하세요?


...
```

`perl` 과 `ruby` 와 같이, Julia 코드를 실행하고 옵션을 지정하는 방법은 다음과 같이 여러가지가 있다.

```
julia [switches] -- [programfile] [args...]
```

> Julia 1.1
>
> In Julia 1.0, the default `--project=@.` option did not search up from the root directory of a Git repository for the `Project.toml` file. From Julia 1.1 forward, it does.

2.1 참고 자료

줄리아 웹사이트의 배우기 페이지에 사용자가 보면 유용한 자료를 엄선하여 모아두었다.

스위치	설명
-v, --version	버전 정보를 표시한다
-h, --help	command-line 옵션(이 메세지)을 표시한다
--project[={<dir>|@.}]	Set <dir> as the home project/environment. The default @. option will search through parent directories until a Project.toml or JuliaProject.toml file is found.
-J, --sysimage <file>	주어진 시스템 이미지 파일로 실행한다
-H, --home <dir>	julia 실행파일의 위치를 지정한다
--startup-file={yes|no}	~/.julia/config/startup.jl 를 불러온다
--handle-signals={yes|no}	Julia의 기본 시그널 핸들러를 켜거나 끈다
--sysimage-native-code={yes|no}	시스템 이미지의 기존 코드 사용/사용하지 않는다
--compiled-modules={yes|no}	모듈의 사전 증분 컴파일을 활성화/비활성화 한다
-e, --eval <expr>	<expr>를 실행만 한다
-E, --print <expr>	<expr>를 실행하고 결과를 표시한다
-L, --load <file>	<file>을 모든 프로세서에 로드한다
-p, --procs {N|auto}	N개의 로컬 worker 프로세스를 추가로 생성한다; auto는 로컬 CPU 스레드 (논리적 코어) 만큼의 worker 프로세스를 생성한다
--machine-file <file>	<file>에 나열된 호스트에서 worker 프로세스를 실행한다
-i	대화형 모드; PEPL을 돌리며 ininteractive()는 true이다
-q, --quiet	깔끔히 시작하기: 배너 없이, REPL 경고도 안 보여준다
--banner={yes|no|auto}	시작 배너 사용/사용하지 않는다
--color={yes|no|auto}	모든 텍스트에 색상을 표시하거나 표시하지 않는다
--history-file={yes|no}	작업내역을 저장하거나 로드한다
--depwarn={yes|no|error}	문법과 함수가 폐기됐다는 경고를 활성화/비활성화 한다 (error는 경고를 에러로 바꾼다)
--warn-overwrite={yes|no}	메소드 오버라이딩 경고를 활성화/비활성화 한다
-C, --cpu-target <target>	<target>까지의 CPU 기능만을 사용한다; 사용 가능한 옵션을 보려면 help로 설정
-O, --optimize={0,1,2,3}	코드 실행시간에 관련된 최적화를 실행한다 (지정되지 않을 경우 2단계 실행, 레벨 이외의 값을 사용할 경우 3단계 실행)
-g, -g <level>	디버그 정보 생성 수준을 활성화/비활성화 합니다 (지정되지 않을 경우 레벨 1, 레벨 이외의 값을 사용할 경우 레벨 2)
--inline={yes|no}	@inline으로 선언된 함수를 덮어쓰는 경우를 포함해서, 인라이닝을 허용할지 결정한다
--check-bounds={yes|no|auto}	배열의 경계 체크를 항상 실행/생략한다 (변수 선언을 무시)

Chapter 3

변수

변수는 값의 이름이다. 나중에 값을 재사용하려고 한다면 변수에 저장하여 활용할 수 있다:

```
# 변수 x에 10을 할당한다.
julia> x = 10
10

# x의 값으로 계산할 수 있다.
julia> x + 1
11

# x의 값을 재할당할 수 있다.
julia> x = 1 + 1
2

# 기존에 저장된 값과 다른 타입을 저장할 수 있다(예시: 문자열).

julia> x = "Hello World!"
"Hello World!"
```

Julia는 유연한 변수 명명법을 가진다. 일단 변수 이름은 대소문자를 구분한다:

```
julia> x = 1.0
1.0

julia> y = -3
-3
```

```
julia> Z = "My string"
"My string"

julia> customary_phrase = "Hello world!"
"Hello world!"

julia> UniversalDeclarationOfHumanRightsStart = "모든 인간은 태어날 때부터 자유로우며 그 존엄과 권리에 있어
↪  동등하다."
"모든 인간은 태어날 때부터 자유로우며 그 존엄과 권리에 있어 동등하다."
```

Julia는 유니코드 기반의 변수를 허용한다:

```
julia> δ = 0.00001
1.0e-5

julia> 안녕하세요 = "Hello"
"Hello"
```

REPL 및 다른 여러 줄리아 편집 환경에서 많은 유니코드 수학 기호를 입력할 수 있다. \ - LaTeX 기호명 - tab을 입력하면 기호로 변환된다. 예를 들어 변수명 δ는 **\delta**-tab, α̂₂는 **\alpha**-tab-**\hat**-tab-**_2**-tab 으로 입력할 수 있다. 특정 기호의 LaTex 기호명이 궁금하다면 REPL에서 ?(help 호출)를 입력하고 원하는 유니코드 기호를 넣어서 확인할 수 있다.

julia는 심지어 내장 상수와 함수를 필요한 경우 변수로 재정의할 수 있다(추천하지 않는 방법이다):

```
julia> pi = 3
3

julia> pi
3

julia> sqrt = 4
4
```

그러나 이미 사용했던 내장 상수나 함수를 다시 정의하려고 하면 다음과 같은 오류를 낸다.

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
```

```
ERROR: cannot assign a value to variable MathConstants.pi from module Main

julia> sqrt(100)
10.0

julia> sqrt = 4
ERROR: cannot assign a value to variable Base.sqrt from module Main
```

##허용하는 변수 이름

변수 이름은 문자(A-Z 또는 a-z), 밑줄(_) 또는 00A0보다 큰 유니코드 코드의 부분 집합으로 시작해야 한다. 특히 유니 코드 문자 범주 Lu/Ll/Lt/Lm/Lo/Nl(문자), Sc/So(통화 및 기타 기호) 및 기타 문자와 유사한 문자(Sm의 부분집합)가 허용된다. 그 다음 문자는 !와 숫자 (0-9와 Nd/No에 포함된 문자) 및 기타 유니코드 코드 포인트: (Mn/Mc/Me/Sk)의 발음 구별 기호 및 기타 수정 표시, 구두점(Pc), 소수(primes) 및 몇 가지 다른 문자)를 포함할 수 있다.

+와 같은 연산자도 유니코드에 포함되지만 예외적으로 파싱된다. 일부 조건에서는 연산자를 변수로 사용할 수 있다. 예를 들어(+)는 더하기 함수를 가리키고, (+) = f로 재할당을 할 수 있다. ⊕와 같은 대부분의 유니코드 중위 연산자(Sm)는 Julia의 중위 연산자로 파싱되며 사용자 정의 메소드(예 :크로네커 곱을 정의하기 위해const ⊗ = kron를 정의)를 사용할 수도 있다. 연산자는 수정 기호, 프라임 기호, 윗첨자/아래첨자를 접두로 붙일 수 있다. 예를 들어 +ˆₐ″는+와 같은 우선 순위를 가진 중위 연산자로 파싱된다.

허용되지 않는 변수 이름은 내장 Keywords뿐이다.

```
julia> else = false
ERROR: syntax: unexpected "else"

julia> try = "No"
ERROR: syntax: unexpected "="
```

일부 유니코드 문자는 동등하게 간주된다. 유니코드가 달라도 문자가 같으면 동일하게 취급된다(Julia는 NFC 표준을 사용함). 예를 들어 유니코드 문자 (U + 025B : 라틴어 소문자 열린 e)와 μ(U + 00B5 : 마이크로 부호)는 이와 형태가 같은 그리스 문자와 동일하게 취급된다.

3.1 문체 규칙

Julia는 변수 명명법이 자유롭지만 다음 규칙을 준수하는 것을 추천한다:

- 변수는 소문자를 사용한다.
- '_'와 같은 문자는 다른 사람을 배려해 되도록 사용하지 않는다.

- 타입과 모듈 이름은 대문자로 시작하고 단어 분리는 밑줄(_)대신 쌍봉낙타 표기법(UpperCamelCase: 모든 단어의 첫글자를 대문자로 표기)을 지향한다.
- **함수**와 **매크로**의 이름에는 밑줄을 넣지 않는다.
- in-place 함수의 이름은 !로 끝난다.

Chapter 4

정수와 부동 소수점 수

정수와 부동 소수점 수는 수치 연산에 있어 기본적인 구성 요소이다. 줄리아 내에서 이와 같은 값의 표현은 숫자 프리미티브(numeric primitives)로 불리운다. 정수와 부동 소수점 표현은 코드에서 바로 그 값을 가지며 수치형 리터럴(numeric literals)로 알려 있다. 예를 들어, `1`은 정수형 리터럴이고, `1.0`은 부동 소수점 수인 리터럴이다; 이들의 메모리 상 바이너리를 객체(object)로 표현하면 숫자 프리미티브이다.

줄리아는 표준 수학 함수에 쓰이는 기본 수치 타입과 완전한 산술 비트 연산자를 폭넓게 제공한다. 이들을 컴퓨터가 지원하는 숫자 타입과 연산에 직접 매핑하여 줄리아가 최적의 연산 자원을 활용할 수 있도록 한다. 추가로 줄리아는 임의 정밀도 산술(Arbitrary Precision Arithmetic)을 소프트웨어적으로 지원하는데, 하드웨어적 표현으로는 효율적으로 구성하기 힘든 아주 정밀한 숫자를 다루며 상대적으로 성능은 다소 줄게 된다.

다음은 줄리아에서 기본적으로 지원하는 타입이다:

- 정수형 타입:

- 부동소수점 타입:

추가적으로 복소수와 유리수는 위에서 언급한 타입에 기초하여 만들어졌다. 모든 기본 수치 타입들은 유연하고, 쉽게 확장이 가능한 type promotion system 덕분에 자유롭게 상호운용이 가능하다.

4.1 정수

정수 리터럴은 다음과 같은 표준적인 방식으로 표한한다:

```
julia> 1
1
```

타입	부호 여부	비트 수	최솟값	최댓값
Int8	☒	8	-2^7	2^7 - 1
UInt8		8	0	2^8 - 1
Int16	☒	16	-2^15	2^15 - 1
UInt16		16	0	2^16 - 1
Int32	☒	32	-2^31	2^31 - 1
UInt32		32	0	2^32 - 1
Int64	☒	64	-2^63	2^63 - 1
UInt64		64	0	2^64 - 1
Int128	☒	128	-2^127	2^127 - 1
UInt128		128	0	2^128 - 1
Bool	N/A	8	false (0)	true (1)

타입	정밀도	비트 수
Float16	half	16
Float32	single	32
Float64	double	64

```
julia> 1234
1234
```

정수 리터럴은 해당 시스템이 32비트 아키텍처인지 혹은 64비트 아키텍처인지에 따라 결정된다:

```
# 32-bit system:
julia> typeof(1)
Int32

# 64-bit system:
julia> typeof(1)
Int64
```

줄리아의 내부변수 Sys.WORD_SIZE는 해당 시스템이 32비트인지 64비트인지 알려주는 역할을 한다:

```
# 32-bit system:
julia> Sys.WORD_SIZE
```

```
32

# 64-bit system:
julia> Sys.WORD_SIZE
64
```

줄리아는 부호가 있는 정수형과 부호가 없는 정수형을 위해 `Int`와 `UInt`라는 타입 또한 정의하고 있다:

```
# 32-bit system:
julia> Int
Int32
julia> UInt
UInt32

# 64-bit system:
julia> Int
Int64
julia> UInt
UInt64
```

32비트로 표현할 수 없지만 64비트로 표현이 가능한 큰 정수형 리터럴은 시스템의 타입과는 상관없이 항상 64비트를 생성한다:

```
# 32-bit or 64-bit system:
julia> typeof(3000000000)
Int64
```

부호가 없는 정수형의 입출력은 항상 `0x`라는 접두어가 붙으며 16진수는 `0-9a-f`범위의 숫자와 문자를 쓴다.(입력할 때, 대문자 `A-F`도 쓸 수 있다.) :

```
julia> 0x1
0x01

julia> typeof(ans)
UInt8

julia> 0x123
0x0123

julia> typeof(ans)
```

```
UInt16

julia> 0x1234567
0x01234567

julia> typeof(ans)
UInt32

julia> 0x123456789abcdef
0x0123456789abcdef

julia> typeof(ans)
UInt64

julia> 0x111122223333444455556666777788888
0x111122223333444455556666777788888

julia> typeof(ans)
UInt128
```

일반적으로 부호가 없는 16진수 정수 리터럴을 쓸 때, 단순히 정수를 표현사기 보다는 사람들은 고정된 바이트 시퀀스(fixed numeric byte sequence)를 표현하기 위해 16진수를 쓰는 경향이 있기 때문에, 위와 같이 부호가 없는 정수형에 16진수 형태를 결합시키도록 하였다.

ans 가 대화형 실행 환경에서 가장 최근에 실행된 표현식의 결과를 나타내었다는 것을 떠올려보면, 위의 줄리아 코드는 다른 환경에서는 제대로 실행이 안될 것이라는 것을 알 수 있다.

줄리아는 2진수와 8진수 리터럴 또한 지원한다:

```
julia> 0b10
0x02

julia> typeof(ans)
UInt8

julia> 0o010
0x08

julia> typeof(ans)
UInt8
```

```
julia> 0x000000000000000001111222233334444
0x000000000000000001111222233334444

julia> typeof(ans)
UInt128
```

As for hexadecimal literals, binary and octal literals produce unsigned integer types. The size of the binary data item is the minimal needed size, if the leading digit of the literal is not `0`. In the case of leading zeros, the size is determined by the minimal needed size for a literal, which has the same length but leading digit `1`. That allows the user to control the size. Values which cannot be stored in `UInt128` cannot be written as such literals.

Binary, octal, and hexadecimal literals may be signed by a `-` immediately preceding the unsigned literal. They produce an unsigned integer of the same size as the unsigned literal would do, with the two's complement of the value:

```
julia> -0x2
0xfe

julia> -0x0002
0xfffe
```

정수형과 같은 기본 수치 타입의 최소값과 최대값은 `typemin`과 `typemax` 함수를 통해 알 수 있다:

```
julia> (typemin(Int32), typemax(Int32))
(-2147483648, 2147483647)

julia> for T in [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
           println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
       end
   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

typemin과 typemax가 제공하는 값들은 항상 매개변수의 타입과 같은 타입을 가진다. (위 예제에서 쓰는 for loops, Strings, Interpolation과 같은 표현들은 아직 소개하지 않은 것들이다. 그러나 약간의 프로그래밍 지식이 있다면 이해하기 별 문제가 없을 것이다.)

오버플로우(Overflow) 동작

줄리아에서는 주어진 타입에서 표현할 수 있는 값을 넘어서게 되면 다음과 같이 주어진 범위를 벗어나지 않는(wraparound) 동작을 보여준다:

```
julia> x = typemax(Int64)
9223372036854775807

julia> x + 1
-9223372036854775808

julia> x + 1 == typemin(Int64)
true
```

위와 같기 때문에, Juali 정수의 연산은 사실 나머지 연산임을 알 수 있다. 오버플로우가 나올 수 있는 프로그램에서는 오버플로우를 명시적으로 체크하는 것이 필수적이다; 그런 경우가 아니라면 Arbitrary Precision Arithmetic에서 BigInt타입을 사용하는 것을 추천한다.

나눗셈 관련 에러들

정수의 나눗셈 연산 (div 함수)는 두 가지 예외적인 경우가 있다; 0으로 나누기, 그리고 컴퓨터가 표현할 수 있는 최소의 음의 정수값(typemin)을 -1로 나누는 것이다. 두 가지 경우 DivideError를 유발한다. 두 나머지 연산 함수(rem과 mod)는 두 번째 매개변수가 0일 때, DivideError를 던진다(throw).

4.2 부동소수점으로 표현되는 실수

부동소수점 리터럴은 필요할 때 E-notation 표준 포맷을 이용하여 표현된다:

```
julia> 1.0
1.0

julia> 1.
1.0

julia> 0.5
```

```
0.5

julia> .5
0.5

julia> -1.23
-1.23

julia> 1e10
1.0e10

julia> 2.5e-4
0.00025
```

위에 나온 결과는 모두 `Float64`타입의 값들이다. `Float32`값들은 `e`대신 `f`를 쓰면 입력할 수 있다:

```
julia> 0.5f0
0.5f0

julia> typeof(ans)
Float32

julia> 2.5f-4
0.00025f0
```

값들은 쉽게 `Float32`타입으로 변환할 수 있다:

```
julia> Float32(-1.5)
-1.5f0

julia> typeof(ans)
Float32
```

16진수로 표현되는 부동소수점 리터럴은 유효하지만, Base-2 이전에 `p`를 사는 경우 `Float64`타입에서만 가능하다:

```
julia> 0x1p0
1.0

julia> 0x1.8p3
```

```
12.0

julia> 0x.4p-1
0.125

julia> typeof(ans)
Float64
```

16비트의 정밀도가 절반인 부동소수점(Half-precision floating-point)(`Float16`)도 지원되지만, 계산을 위해 `Float32`를 사용한 소프트웨어로 구현되어있다.

```
julia> sizeof(Float16(4.))
2

julia> 2*Float16(4.)
Float16(8.0)
```

밑줄_는 수 구분자로 쓰일 수 있다:

```
julia> 10_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
(10000, 5.0e-9, 0xdeadbeef, 0xb2)
```

실수로서의 숫자 0

부동 소수점은 양수 0과 음수 0으로 불리는 두 개의 0을 가진다. 그 둘은 값은 0으로써 같지만, 다음과 같이 `bitstring` 함수를 사용하면, 바이너리로 표기했을 때와 차이를 알 수 있다:

```
julia> 0.0 == -0.0
true

julia> bitstring(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

julia> bitstring(-0.0)
"1000000000000000000000000000000000000000000000000000000000000000"
```

특별한 부동 소수점 값들

다음은 실수에는 포함되지는 않은 세 종류의 특정 표준 부동 소수점이 있다:

Float16	Float32	Float64	이름	설명
Inf16	Inf32	Inf	positive infinity	모든 유한한 부동 소수점 실수보다 큰 값
-Inf16	-Inf32	-Inf	negative infinity	모든 유한한 부동 소수점 실수보다 작은 값
NaN16	NaN32	NaN	not a number	어떤 부동 소수점 실수와도 같지 않은 값

이와 같은 유한하지 않은 부동 소수점 값들이 서로와 다른 실수에 대해서 순서를 매길 때에는 비교 연산을 참고하길 바란다. IEEE 754 standard에 따르면, 위에서의 부동 소수점 실수들은 어떤 산술 연산에 의한 결과임을 알 수 있다:

```
julia> 1/Inf
0.0

julia> 1/0
Inf

julia> -5/0
-Inf

julia> 0.000001/0
Inf

julia> 0/0
NaN

julia> 500 + Inf
Inf

julia> 500 - Inf
-Inf

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> Inf * Inf
Inf
```

```
julia> Inf / Inf
NaN

julia> 0 * Inf
NaN
```

typemin과 typemax 함수는 부동 소수점 타입에도 적용이 가능하다:

```
julia> (typemin(Float16),typemax(Float16))
(-Inf16, Inf16)

julia> (typemin(Float32),typemax(Float32))
(-Inf32, Inf32)

julia> (typemin(Float64),typemax(Float64))
(-Inf, Inf)
```

계산기 입실론(Machine epsilon)

대부분 실수들은 부동 소수점 형태로는 정확하게 표현할 수 없다. 그리고 현재로서는 많은 경우 두 인접한 부동 소수점으로 표현 가능한 실수가 얼만큼 떨어져 있는지 알 필요가 있다. 따라서 이를 위해 계산기 입실론(machine epsilon)이라는 개념이 도입하게 되었다.

줄리아는 eps라는 것을 제공한다. 이는 `1.0`과 `1.0` 다음으로 큰 표현 가능한 부동 소수점 값과의 거리를 말한다:

```
julia> eps(Float32)
1.1920929f-7

julia> eps(Float64)
2.220446049250313e-16

julia> eps() # same as eps(Float64)
2.220446049250313e-16
```

위 코드에서 나오는 값들은 Float64와 Float64값 중에서 바이너리로 표기했을 때, `2.0^-23`과 `2.0^-52`를 각각 나타낸다. eps 함수는 부동 소수점 실수를 매개변수로 받을 수도 있는데, 이 때는 `1.0`이 아니라 주어진 값과 주어진 값 바로 옆에서 주어진 값과의 거리를 반환한다. 그 말은 `epx(x)`의 반환값은 `x`와 같은 타입이고, `x+eps(x)`는 `x`보다 큰 `x`바로 옆에 있는 포현 가능한 부동 소수점 실수를 뜻한다:

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(1000.)
1.1368683772161603e-13

julia> eps(1e-27)
1.793662034335766e-43

julia> eps(0.0)
5.0e-324
```

두 인접하면서 표현 가능한 부동 소수점 실수들은 상수가 아니지만, 작은 값에서는 작은 값을 지니고, 큰 값들에서는 큰 값을 나타낸다. 다른 말로 하면, 표현 가능한 부동 소수점 실수들은 실수축 상에서 0에 근접할 때 가장 밀집되어있고, 0에서 멀어질수록 점점 드물다. 정의에 의하면, `eps(1,0)`은 `eps(Float64)`와 같은데, 그 이유는 `1.0`은 64비트 부동 소수점 실수이기 때문이다.

또한 줄리아는 `nextfloat`과 `prevfloat` 함수를 제공하는데, 이는 표현 가능한 부동 소수점 실수 중에서 주어진 실수 바로 옆에있는 크거나 작은 수를 반환한다:

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0

julia> bitstring(prevfloat(x))
"00111111100111111111111111111111"

julia> bitstring(x)
"00111111101000000000000000000000"

julia> bitstring(nextfloat(x))
"00111111101000000000000000000001"
```

위의 예제는 서로 이웃한 표현 가능한 부동 소수점 수는 바이너리 정수 표기법을 가질 수 있다는 기본적인 원리를 새삼 일깨워준다.

라운딩 모드

만약 어떤 숫자가 정확한 부동 소수점 표현을 가지고 있지 않다면, 그 수는 반드시 어떤 표현 가능한 값으로 라운딩 되어야 한다. IEEE 754 표준에 나오는 라운딩 모드로 라운딩 방식을 바꿀 수 있다.

기본 라운딩 모드는 항상 `RoundNearest`이다. 이는 가장 근접한 표현 가능한 값으로 라운딩 하지만, 만약 두 표현 값 중간에 주어진 값이 걸쳐 있으면 가수부 값 중 짝수(바이너리 임으로 0)로 라운딩 하는 모드이다.

부동 소수점 실수에 대해서 더 읽으면 좋은 문서들

부동 소수점 연산은 많은 미묘한 것들을 수반하고 있기 때문에 저수준(low-level) 구현에 익숙하지 않은 유저들은 당황할 수도 있다. 그러나 그 미묘한 점들은 과학적 연산과 관련된 많은 책들에서 잘 설명되고 있고, 아래에 나열하는 참고문헌도 참고하면 좋을 것이다:

- 부동 소수점과 관련해서 가장 확실한 가이드는 IEEE 754-2008 Standard이지만, 유료이다.
- 부동 소수점이 어떻게 표현되는지에 대한 간략하면서도 명쾌한 설명은 John D. Cook's 블로그 글를 참고하면 된다. 같은 주제에 관하여 이와 더불어서 그의 소개글은 부동 소수점이 실수의 이상적인 추상화와 다름으로써 생기는 몇가지 문제에 대해서도 다루고 있다.
- Bruce Dawson의 series of blog posts on floating-point numbers도 추천하는 바이다.
- 상급자들은 부동 소수점의 내부 구현에 관한 이야기들과 부동 소수점 연산을 할 때 맞닥뜨릴 수 있는 수치적인 정확도에 관한 문제들에 대해서는 David Goldberg의 논문 What Every Computer Scientist Should Know About Floating-Point Arithmetic를 참고하는 것이 좋다.
- 특별히 부동 소수점의 역사, 근거 및 문제점에 대한 훨씬 더 자세한 문서화, 수치 컴퓨팅에서의 토론은 일반적으로 "부동 소수점의 아버지"로 알려진 William Kahan의 collected writings을 참조하세요. 관심이 더 생긴다면 An Interview with the Old Man of Floating-Point를 읽기 바란다.

4.3 임의 정밀도 연산

임의 정밀도의 정수와 부동 소수점들의 연산을 위해, 줄리아는 GNU Multiple Precision Arithmetic Library (GMP)와 GNU MPFR Library을 각각 래빙(wrapping)하였다. `BigInt`와 `BigFloat`타입은 줄리아에서 각각 임의 정밀도의 정수와 부동 소수점을 다루기 위해 사용되고 있다.

기본 수치 타입으로부터 임의 정밀도 정수와 부동 소수점 타입을 만들기 위해 생성자가 존재하며, `parse`는 `AbstractString`들로부터 임의 정밀도 타입을 만들 수 있게 해준다. 한번 임의 정밀도 타입이 만들어지면, type promotion and conversion mechanism덕분에 자유롭게 다른 수치타입과 연산을 수행할 수 있다:

```
julia> BigInt(typemax(Int64)) + 1
9223372036854775808
```

```
julia> parse(BigInt, "123456789012345678901234567890") + 1
123456789012345678901234567891

julia> parse(BigFloat, "1.23456789012345678901")
1.234567890123456789010000000000000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.459565876494606882133333333333333333333333333333333333333333333333333333333333344e+19

julia> factorial(BigInt(40))
815915283247897734345611269596115894272000000000
```

그러나, 기본 타입과 BigInt/BigFloat간의 묵시적 형 변환(type promotion)은 자동으로 이루어지지 않고, 반드시 명시적으로 처리되어야 한다.

```
julia> x = typemin(Int64)
-9223372036854775808

julia> x = x - 1
9223372036854775807

julia> typeof(x)
Int64

julia> y = BigInt(typemin(Int64))
-9223372036854775808

julia> y = y - 1
-9223372036854775809

julia> typeof(y)
BigInt
```

BigFloat타입에서 기본 정밀도(가수부의 비트수)와 라운딩 모드는 setprecision와 setrounding를 호출함으로써 변경할 수 있으며, 한 번 호출된 이후에는 그 설정이 계속 유지 된다. 특정 블럭의 코드에서만 정밀도와 라운딩을 변경하기 위해서는 do블럭의 코드에서와 같은 함수를 호출한다:

```
julia> setrounding(BigFloat, RoundUp) do
           BigFloat(1) + parse(BigFloat, "0.1")
```

```
       end
1.100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000003

julia> setrounding(BigFloat, RoundDown) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.099999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999986

julia> setprecision(40) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.1000000000004
```

4.4 수치형 리터럴 계수

보통의 수학 식과 표현식을 깔끔하게 표현하기 위해서, 줄리아는 변수가 수치형 리터럴 바로 다음에 있으면 둘 사이의 관계가 곱셈임을 가정한다. 이는 다항식의 표현을 더욱 깔끔하게 만든다:

```
julia> x = 3
3

julia> 2x^2 - 3x + 1
10

julia> 1.5x^2 - .5x + 1
13.0
```

이는 지수함수의 표현도 매우 아름답게 만들 수 있다:

```
julia> 2^2x
64
```

The precedence of numeric literal coefficients is slightly lower than that of unary operators such as negation. So `-2x` is parsed as `(-2) * x` and `√2x` is parsed as `(√2) * x`. However, numeric literal coefficients parse similarly to unary operators when combined with exponentiation. For example `2^3x` is parsed as `2^(3x)`, and `2x^3` is parsed as `2*(x^3)`.

수치형 리터럴은 괄호가 있는 식에서도 계수(coeffiients)로 작동할 수 있다:

```
julia> 2(x-1)^2 - 3(x-1) + 1
3
```

!!! 중요 : 내재적(implicit)으로 사용된 수치형 리터럴 계수의 선행은 다른 곱셈(*)과 나눗셈(/,\, //) 같은 이진 연산자보다 높습니다. 예를들어 `1/2im1`은 `-0.5im`과 같고, `6//2(2+1)`는 `1//1`과 같습니다.

게다가 괄호 표현식은 변수 또한 계수로 생각하여, 곱셈기호 없이도 변수들 간의 곱으로 식을 표현할 수도 있다:

```
julia> (x-1)x
6
```

그러나 두 괄호식을 병치하거나 괄호식 앞에 변수를 두는 경우는 계수로 사용할 수 없다:

```
julia> (x-1)(x+1)
ERROR: MethodError: objects of type Int64 are not callable

julia> x(x+1)
ERROR: MethodError: objects of type Int64 are not callable
```

두 표현식은 함수로써 인식된다. 괄호앞에 붙는 수치형 리터럴이 아닌 표현식들은 모두 함수와 함수의 매개변수로 인식된다(자세한 설명을 위해서는 Functions를 참고하도록 하자). 그래서 두 가지 경우 모두 왼쪽에 있는 값이 함수가 아님을 알려주는 에러가 발생한다.

위에서 언급한 문법적 강화효과는 수학식을 작성할 때 생기는 시각적 공해를 줄일 수 있도록 해준다. 단지 한 가지 알아야 할 점은 수치형 계수와 이에 곱해지는 변수 혹은 괄호식 등 사이에는 빈칸이 있어서는 안된다.

문법적 충돌

리터럴 계수를 병치하는 문법은 16진수 정수 리터럴과 부동 소수점의 공학적 표현이라는 두 수치형 리터럴 문법과 충돌이 생길 수 있다. 다음은 문법적 충돌이 발생하는 예이다:

- 16진수 리터럴 표현식 `0xff`는 수치형 리터럴 `0`과 변수 `xff`의 곱셈으로 해석될 수 있다.
- 부동 소수점 리터럴 표현식 `1e10`은 수치형 리터럴 `1`이 변수 `e10`에 곱해지는 걸로 해석될 수 있고 이는 `e`가 아닌 `E`를 쓸 때에도 마찬가지이다.
- The 32-bit floating-point literal expression `1.5f22` could be interpreted as the numeric literal `1.5` multiplied by the variable `f22`.

이와 같은 경우에, 우리는 수치형 리터러를 해석하는데 있어서 다음과 같은 방식으로 모호함을 해결했다:

- `0x`로 시작하는 표현식은 항상 16진수 리터럴이다.

- 수치형 리터럴으로 시작하는 표현식에서 수치형 리터럴 다음에 `e`또는 `E`가 뒤따라오면 항상 부동소수점 리터럴이다.
- Expressions starting with a numeric literal followed by `f` are always 32-bit floating-point literals.

Unlike `E`, which is equivalent to `e` in numeric literals for historical reasons, `F` is just another letter and does not behave like `f` in numeric literals. Hence, expressions starting with a numeric literal followed by `F` are interpreted as the numerical literal multiplied by a variable, which means that, for example, `1.5F22` is equal to `1.5 * F22`.

4.5 리터럴 0과 1

줄리아는 어떤 특정한 타입이나 주어진 변수의 타입에 따라 리터럴 0이나 1을 리턴하는 함수를 제공한다.

함수	설명
zero(x)	x타입이나 변수 x의 타입의 리터럴 0
one(x)	x타입이나 변수 x의 타입의 리터럴 1

위 함수들은 비교 연산에서 불필요한 type conversion에 의한 성능저하를 줄일 때 유용하다.

Examples:

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
0.0

julia> one(Int32)
1

julia> one(BigFloat)
1.0
```

Chapter 5

산술 연산과 기본 함수

Julia가 제공하는 숫자형 타입은 산술 연산자와 비트 연산자, 다양한 수학적 함수를 지원한다.

5.1 산술 연산자

다음 산술 연산자들은 모든 숫자형 타입에 사용할 수 있다:

표현식	이름	설명
`+x`	단항 덧셈	항등 연산
`-x`	단항 뺄셈	덧셈의 역원 반환
`x + y`	덧셈	일반적인 덧셈
`x - y`	곱셈	일반적인 뺄셈
`x * y`	곱셈	일반적인 곱셈
`x / y`	나눗셈	일반적인 나눗셈
`x ÷ y`	정수 나눗셈(몫)	`x / y`의 몫 반환
`x \ y`	역 나눗셈	`y / x`와 동일
`x ^ y`	제곱	`x` 의 `y`제곱을 반환
`x % y`	나머지	`rem(x,y)`와 동일(나머지를 반환)

`Bool` 타입에 대한 부정 연산도 가능하다:

표현식	이름	설명
`!x`	부정 연산	`true`를 `false`로 바꾸거나 혹은 그 반대

줄리아의 타입 치환 시스템은 산술 연산이 자연스럽게 작동하게 한다. 자세한 것은 Conversion and Promotion을 참고하라.

산술 연산자를 활용한 간단한 예제다:

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(일반적으로 근처 다른 연산자보다 먼저 적용되는 경우 간격을 밀접하게 두는 경우가 있다. 예를 들어 x를 음수로 먼저 변환하고 2를 반환하는 코드는 편의상 `-x + 2`로 쓴다)

5.2 비트 연산자

비트 연산자는 모든 기본 정수형 타입을 지원한다:

Expression	Name
`~x`	비트 부정
`x & y`	비트 and 연산
`x | y`	비트 or 연산
`x ⊻ y`	비트 xor 연산 (exclusive or)
`x >>> y`	logical shift right
`x >> y`	arithmetic shift right
`x << y`	logical/arithmetic shift left

비트 연산자를 활용한 간단한 예제다:

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251
```

```
julia> 123 ⊻ 234
145

julia> xor(123, 234)
145

julia> ~UInt32(123)
0xffffff84

julia> ~UInt8(123)
0x84
```

5.3 업데이트 연산자

산술 연산자와 비트 연산자는 그에 대응하는 업데이트 연산자가 있습니다. 업데이트 연산자는 변수의 값과 새롭게 제시된 피연산자로 계산한 후 결과를 다시 해당 변수에 저장합니다. 업데이트 연산자는 기존 연산자 기호 우측에 =를 붙임으로써 만들 수 있습니다. 예를 들어 `x += 3`는 `x = x + 3`와 같은 의미가 됩니다:

```
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

각 산술/비트 연산자에 대응하는 업데이트 연산자는 아래와 같습니다:

```
+=  -=  *=  /=  \=  ÷=  %=  ^=  &=  |=  ⊻=  >>>=  >>=  <<=
```

> Note
>
> Julia는 상황에 따라 타입을 바꾸기 때문에, 업데이트 연산자가 변수의 타입을 바꿀 수 있다.
>
> ```
> julia> x = 0x01; typeof(x)
> UInt8
>
> julia> x *= 2 # Same as x = x * 2
> ```

```
2


julia> typeof(x)
Int64
```

5.4 배열에서의 연산("dot" 연산자)

^와 같은 모든 이진 연산자는 배열의 원소별 연산을 위한 "dot" 연산자 .^가 있다. 따라서 [1,2,3]의 모든 원소를 세제곱 하고 싶다면 [1,2,3] ^ 3이 아니라 [1,2,3] .^ 3로 작성해야 한다. !같은 단항 연산자도 사용할 수 있다(.!).

```
julia> [1,2,3] .^ 3
3-element Array{Int64,1}:
  1
  8
 27


julia> .![true,false,true]
3-element BitArray{1}:
 0
 1
 0
```

보다 구체적으로, a .^ b는 (^).(a,b)로 해석되고, 여기서 .은 broadcast 연산을 한다: broadcast 연산은 배열과 스칼라, 배열과 배열(모양이 달라도 됨)을 원소별 연산이 가능하게 같은 모양의 배열로 "적절히" 바꿔준다(예를 들어 row 벡터와 column 백터가 들어오면 행렬을 생성한다). 또한 "dot" 연산자는 근처 다른 "dot" 연산자와 결합하여 반복문을 한번만 돌리도록 설계되었다. 만약 2 .* A.^2 .+ sin.(A)(혹은 @. macro)을 사용하여 @. 2A^2 + sin(A)를 계산한다면, Julia는 A의 모든 원소에 대해 2a^2 + sin(a)를 계산한다. f.(g.(x))같은 nested dot 호출도 이런 최적화가 일어나기 때문에 x .+ 3 .* x.^2와 (+).(x, (*).(3, (^).(x, 2)))같은 함수 꼴로 사용해도 성능상 차이가 발생하지 않는다.

나아가서, in-place 융합된 대입 연산 .=에 대해 a .+= b (or @. a += b)와 같은 "dot" 업데이트 연산자들은 a .= a .+ b로 구문분석된다(are parsed).

(dot 문법 문서을 참고하라).

dot 연산자는 사용자 정의 연산자에서도 활용할 수 있다. 예를 들어 ⊗(A,B) = kron(A,B)를 정의했다면 [A,B] .⊗ [C,D]은 [A⊗C, B⊗D]를 계산한다.

dot 연산자를 숫자형 리터럴과 혼용하는 것은 해석의 모호성을 야기할 수 있다. 예를 들어 1.+x은 1. + x인지 1 .+ x인지 확실하지 않다. 따라서 이런 문법은 지원하지 않으며, 불가피하게 사용할 시 여백으로 문법을 명확히 해야한다.

5.5 비교 연산

모든 기본 숫자형 타입은 비교연산을 지원한다(복소수 예외):

연산자	설명
==	상등
!=, ≠	상등 부정
<	작다
<=, ≤	작거나 같다
>	크다
>=, ≥	크거나 같다

아래 예제로 사용법을 볼 수 있다:

```
julia> 1 == 1
true

julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false

julia> 1 >= 1.0
true

julia> -1 <= 1
true

julia> -1 <= -1
```

```
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

정수에서 비교연산은 같은 위치의 비트를 비교하는 방식으로 이뤄진다. 반면 실수는 IEEE 754 standard의 규칙에 따라 비교한다:

- 유한한 수는 일반적인 방식으로 이뤄진다.
- +0과 -0은 서로 같다.
- `Inf` 는 `NaN`와 자신을 제외한 수보다 크고, 자기 자신과는 같다.
- `Inf` 는 `NaN`와 자신을 제외한 수보다 작고, 자기 자신과는 같다.
- `NaN` 는 자신을 포함한 그 어떤 수와 같지 않고, 크지도 않고, 작지도 않다.

마지막 규칙은 다른 규칙보다 극단적이라, 실제 계산에서 예상치 못한 결과를 야기할 수 있다:

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

이러한 문제는 특히 배열을 다룰 때 골머리를 썩게 할 것이다:

```
julia> [1 NaN] == [1 NaN]
false
```

함수	반환값이 참인 조건
`isequal(x, y)`	x와 y 가 같은 때
`isfinite(x)`	x가 유한한 수일 대
`isinf(x)`	x가 무한한 수일 때
`isnan(x)`	x가 숫자가 아닐 때

Julia는 해시값처럼 특수한 값에도 비교연산을 사용할 수 있도록 함수를 지원한다:

`isequal`에서 `NaN`이 서로 같다고 나온다:

```
julia> isequal(NaN, NaN)
true

julia> isequal([1 NaN], [1 NaN])
true

julia> isequal(NaN, NaN32)
true
```

`isequal`은 +0과 -0을 구분할 때도 사용할 수 있다:

```
julia> -0.0 == 0.0
true

julia> isequal(-0.0, 0.0)
false
```

정수의 signed나 unsigned 혹은 실수 사이의 비교연산은 까다롭다. Julia는 타입 충돌 없이 이런 것들이 잘 작동하게 보장한다.

서로 다른 타입에서 `isequal`을 사용하면 `==`을 호출하게 되어있다. 당신이 자신만의 타입에서 동일성을 정의하고 싶다면 `==` method를 정의하면 된다. 여기에 `hash` method도 정의하면 `isequal(x,y)`은 `hash(x) == hash(y)`을 반환한다.

비교연산 이어쓰기

대부분의 언어가 지원하지 않지만, Python의 비교연산 문법처럼 비교연산을 이어쓸 수 있다:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

비교연산 이어쓰기는 코드 구성을 깔끔하게 한다. 비교연산 이어쓰기는 `&&`를 사용하여 연산을 한 것과 똑같이 작용하고, 원소별 연산에서는 `&`을 사용한 것과 동일하다. 쉽게 말하면 우리가 수학적으로 예상한 것과 똑같이 나온다는 것이다. 그 예로 `0 .< A .< 1`는 각 원소가 0과 1 사이에 있는지에 대한 참/거짓을 행렬로 반환한다.

Note the evaluation behavior of chained comparisons:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
1
3
true

julia> v(1) > v(2) <= v(3)
2
1
false
```

첫번째 결과에서 중간값이 한번만 계산됨을 확인할 수 있다. 이를 통해 `v(1) < v(2) && v(2) <= v(3)`로 계산했을 때보다 적은 계산량을 가지고, 비교연산 이어쓰기에서는 기존 프로그래밍 언어와 달리 계산 순서는 미리 예측할 수 없다는 걸 확인할 수 있다. 따라서 비교연산 이어쓰기에서는 계산 순서가 중요한 연산(예시: 입출력)을 하지 말자. 이런 부작용을 감안하고 써야한다면 `&&` 연산자를 활용하자. (Short-Circuit Evaluation을 참고하라).

기본 함수

Julia는 수치 계산을 위한 함수와 연산자를 전폭적으로 지원한다. 이런 연산은 서로 다른 타입의 숫자(정수, 실수, 유리수 등)가 충돌 없이 수학적인 결과와 맞아 떨어지게끔 되어있다.

이런 함수 모두 dot 문법을 지원한다. 예를 들어 `sin.(A)`는 array의 모든 `A`의 원소의 sin값을 구한다.

5.6 연산자 우선순위와 결합성

아래 표는 높은 우선 순위부터 낮은 우선순위별로 연산자를 나열하고, 연산자 결합성을 확인할 수 있다:

[1] 단항연산자 `+` 와 `-`를 연속해서 사용하는 경우, 업데이트 연산자(`++`)와 구별하기 위해 괄호를 명시적으로 사용해야 한다. 다른 단항 연산자와 같이 사용하는 경우엔 right-associativity 규칙에 따라 구문을 분석한다(예시: √√-a를 √(√(-a))로 분석).

[2] The operators `+`, `++` and `*` are non-associative. `a + b + c` is parsed as `+(a, b, c)` not `+(+(a, b), c)`. However, the fallback methods for `+(a, b, c, d...)` and `*(a, b, c, d...)` both default to left-associative evaluation.

분류	연산자	결합성
문법	`.` followed by `::`	왼쪽
제곱	`^`	오른쪽
단항 연산	`+ - √`	오른쪽[1]
Bitshifts	`<< >> >>>`	왼쪽
분수	`//`	왼쪽
곱셈, 나눗셈	`* / % & \ ÷`	왼쪽[2]
덧셈, 뺄셈	`+ - ¦ ⊻`	왼쪽[2]
문법	`: ..`	왼쪽
문법	`¦>`	왼쪽
문법	`<¦`	오른쪽
비교 연산	`> < >= <= == === != !== <:`	결합성 없음
제어 흐름	`&&` followed by `¦¦` followed by `?`	오른쪽
Pair	`=>`	오른쪽
할당	`= += -= *= /= //= \= ^= ÷= %= ¦= &= ⊻= <<= >>= >>>=`	오른쪽

모든 Julia 연산자의 우선순위 목록을 보고 싶다면, 다음 파일의 최상단 코드를 참고하라: src/julia-parser.scm

`Base.operator_precedence`을 통해서도 우선순위를 확인할 수 있다. 반환값이 높을수록 더 우선순위가 더 높다:

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*), Base.operator_precedence(:.)
(11, 13, 17)

julia> Base.operator_precedence(:sin), Base.operator_precedence(:+=), Base.operator_precedence(:(=))  # (Note the
↪  necessary parens on `:(=)`)
(0, 1, 1)
```

연산자 결합성 확인은 `Base.operator_associativity`로 확인할 수 있다:

```
julia> Base.operator_associativity(:-), Base.operator_associativity(:+), Base.operator_associativity(:^)
(:left, :none, :right)

julia> Base.operator_associativity(:⊗), Base.operator_associativity(:sin), Base.operator_associativity(:→)
(:left, :none, :right)
```

`:sin`의 경우 우선순위가 `0`임을 확인할 수 있는데, `0`은 최하위 우선순위가 아니라 유효하지 않은 연산자를 나타낸다. 이와 비슷한 이유로 이런 연산자는 연산자 결합성이 `:none`임을 볼 수 있다.

5.7 Numerical Conversions

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions.

- 표기법 `T(x)` 또는 `convert(T,x)`는 `x`를 type `T`의 값으로 변환한다.
  - 만약 `T`가 부동 소숫점 type이면, 결과값은 표현할 수 있는 가장 가까운 값으로 나타내며, 이는 양 혹은 음의 무한대도 될 수 있다.
  - 만약 `T`가 정수 type이면, `x`를 `T` type으로 나타낼 수 없을 때, `InexactError`가 발생한다.
- `x % T`는 정수 `x`를 법 `2^n`에 대해 합동(congruent to `x` modulo `2^n`)인, type `T`의 정수값으로 변환한다(converts an integer `x` to a value of integer type `T`). 여기서 `n`은 `T` 안의 비트 수이다. In other words, the binary representation is truncated to fit.
- The Rounding functions take a type `T` as an optional argument. For example, `round(Int,x)` is a shorthand for `Int(round(x))`.

The following examples show the different forms.

```
julia> Int8(127)
127

julia> Int8(128)
ERROR: InexactError: trunc(Int8, 128)
Stacktrace:
[...]

julia> Int8(127.0)
127

julia> Int8(3.14)
ERROR: InexactError: Int8(3.14)
Stacktrace:
[...]

julia> Int8(128.0)
```

```
ERROR: InexactError: Int8(128.0)
Stacktrace:
[...]

julia> 127 % Int8
127

julia> 128 % Int8
-128

julia> round(Int8,127.4)
127

julia> round(Int8,127.6)
ERROR: InexactError: trunc(Int8, 128.0)
Stacktrace:
[...]
```

See Conversion and Promotion for how to define your own conversions and promotions.

Rounding 함수

함수	설명	반환값
round(x)	round x to the nearest integer	typeof(x)
round(T, x)	round x to the nearest integer	T
floor(x)	round x towards -Inf	typeof(x)
floor(T, x)	round x towards -Inf	T
ceil(x)	round x towards +Inf	typeof(x)
ceil(T, x)	round x towards +Inf	T
trunc(x)	round x towards zero	typeof(x)
trunc(T, x)	round x towards zero	T

함수	설명
`div(x,y)`, `x÷y`	truncated division; quotient rounded towards zero
`fld(x,y)`	floored division; quotient rounded towards `-Inf`
`cld(x,y)`	ceiling division; quotient rounded towards `+Inf`
`rem(x,y)`	remainder; satisfies `x == div(x,y)*y + rem(x,y)`; sign matches `x`
`mod(x,y)`	modulus; satisfies `x == fld(x,y)*y + mod(x,y)`; sign matches `y`
`mod1(x,y)`	`mod` with offset 1; returns `r∈(0,y]` for `y>0` or `r∈[y,0)` for `y<0`, where `mod(r, y) == mod(x, y)`
`mod2pi(x)`	modulus with respect to 2pi; `0 <= mod2pi(x)  < 2pi`
`divrem(x,y)`	returns `(div(x,y),rem(x,y))`
`fldmod(x,y)`	returns `(fld(x,y),mod(x,y))`
`gcd(x,y...)`	greatest positive common divisor of `x`, `y`,...
`lcm(x,y...)`	least positive common multiple of `x`, `y`,...

함수	설명
`abs(x)`	`x`의 절댓값
`abs2(x)`	`x`절댓값의 제곱
`sign(x)`	`x`의 부호. -1, 0, 혹은 +1를 반환
`signbit(x)`	sign bit가 1인지(true) 혹은 0인지(false)인지 반환
`copysign(x,y)`	a value with the magnitude of `x` and the sign of `y`
`flipsign(x,y)`	a value with the magnitude of `x` and the sign of `x*y`

나눗셈 함수

부호 함수와 절댓값 함수

지수, 로그, 루트 함수

For an overview of why functions like `hypot`, `expm1`, and `log1p` are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: expm1, log1p, erfc, and hypot.

삼각 함수와 쌍곡선 함수

Julia는 모든 삼각 함수와 쌍곡선 함수를 지원한다:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
```

함수	설명
`sqrt(x)`, √x	square root of `x`
`cbrt(x)`, ∛x	cube root of `x`
`hypot(x,y)`	hypotenuse of right-angled triangle with other sides of length `x` and `y`
`exp(x)`	natural exponential function at `x`
`expm1(x)`	accurate `exp(x)-1` for `x` near zero
`ldexp(x,n)`	`x*2^n` computed efficiently for integer values of `n`
`log(x)`	natural logarithm of `x`
`log(b,x)`	base `b` logarithm of `x`
`log2(x)`	base 2 logarithm of `x`
`log10(x)`	base 10 logarithm of `x`
`log1p(x)`	accurate `log(1+x)` for `x` near zero
`exponent(x)`	binary exponent of `x`
`significand(x)`	binary significand (a.k.a. mantissa) of a floating-point number `x`

```
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

이 함수들은 인자를 하나만 받지만, 예외적으로 `atan`는 2개를 받을 수 있으며 이는 `atan2`에 대응한다.

추가로 `sinpi(x)`와 `cospi(x)`는 `sin(pi*x)`, `cos(pi*x)`와 결과는 비슷지만 더 정확한 결과를 산출할 수 있다.

삼각 함수 단위에 호도법(radian)대신 도(°)를 사용하려면 접미사 `d`를 붙인다. 예를 들어 `sind(x)`는 x^0의 sin값을 구한다. 아래는 접미사 `d`를 사용한 모든 삼각 함수를 나열했다:

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

특수 함수

이외에도 다양한 수치 계산용 함수를 패키지로 받을 수 있다 SpecialFunctions.jl.

Chapter 6

복소수와 유리수

Julia에서는 복소수와 유리수를 표현할 수 있고 산술 연산과 기본 함수를 모두 지원한다. Conversion and Promotion으로 연산 결과가 수학적으로 예측한 것과 최대한 비슷하게 나오게 했다.

6.1 복소수

전역 상수 `im`는 -1의 제곱근으로, 복소수 i에 해당한다. (수학자는 이를 `i`로 쓰고 공학자는 `j`로 쓰지만, 이 문자들은 인덱싱할 때 자주 사용하므로 전역 상수의 이름으로 채택되지 않았다.) Julia는 계수와 변수 사이에 곱셈 기호를 생략하는 것을 허용하기 때문에 수학적 표기법을 그대로 사용할 수 있다:

```
julia> 1+2im
1 + 2im
```

복소수는 산술 연산을 지원한다:

```
julia> (1 + 2im)*(2 - 3im)
8 + 1im

julia> (1 + 2im)/(1 - 2im)
-0.6 + 0.8im

julia> (1 + 2im) + (1 - 2im)
2 + 0im

julia> (-3 + 2im) - (5 - 1im)
-8 + 3im
```

```
julia> (-1 + 2im)^2
-3 - 4im

julia> (-1 + 2im)^2.5
2.729624464784009 - 6.9606644595719im

julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im

julia> 3(2 - 5im)
6 - 15im

julia> 3(2 - 5im)^2
-63 - 60im

julia> 3(2 - 5im)^-1.0
0.20689655172413796 + 0.5172413793103449im
```

타입 자동 치환을 지원하기 때문에 계산 결과가 수학적으로 예상한 것과 동일하다:

```
julia> 2(1 - 1im)
2 - 2im

julia> (2 + 3im) - 1
1 + 3im

julia> (1 + 2im) + 0.5
1.5 + 2.0im

julia> (2 + 3im) - 0.5im
2.0 + 2.5im

julia> 0.75(1 + 2im)
0.75 + 1.5im

julia> (2 + 3im) / 2
1.0 + 1.5im

julia> (1 - 3im) / (2 + 2im)
-0.5 - 1.0im
```

```
julia> 2im^2
-2 + 0im

julia> 1 + 3/4im
1.0 - 0.75im
```

리터럴 계수가 나눗셈보다 중요도가 높으므로 `3/4im == 3/(4*im) == -(3/4*im)`가 되는 걸 볼 수 있다.

복소수를 다루기 위한 기본 함수가 제공된다:

```
julia> z = 1 + 2im
1 + 2im

julia> real(1 + 2im) # real part of z
1

julia> imag(1 + 2im) # imaginary part of z
2

julia> conj(1 + 2im) # complex conjugate of z
1 - 2im

julia> abs(1 + 2im) # absolute value of z
2.23606797749979

julia> abs2(1 + 2im) # squared absolute value
5

julia> angle(1 + 2im) # phase angle in radians
1.1071487177940904
```

여기서 `abs`는 일반적으로 아는 복소수의 절댓값을 반환하고, `abs2`는 복소수 절댓값의 제곱값, `angle`은 복소수의 각도를 라디안으로 반환한다.

기본 함수는 복소수에서 잘 정의되어 있다:

```
julia> sqrt(1im)
0.7071067811865476 + 0.7071067811865475im
```

```
julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991518im

julia> exp(1 + 2im)
-1.1312043837568135 + 2.4717266720048188im

julia> sinh(1 + 2im)
-0.4890562590412937 + 1.4031192506220405im
```

수리 계산을 하는 함수에 실수 인자가 들어오면 실수를 반환하고 복소수가 들어오면 복소수를 반환한다. 이런 특성 때문에 `sqrt`는 `-1`이 들어올 때와 `-1 + 0im` 이 들어올 때 `-1 == -1 + 0im` 이어도 결과가 다르게 나오는 걸 확인할 수 있다.

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

변수에 저장된 값으로 복소수를 만들 때는 리터럴 계수 표현법대신 직접적으로 곱셈을 써줘야 한다:

```
julia> a = 1; b = 2; a + b*im
1 + 2im
```

하지만 위처럼 복소수를 만드는 것은 추천하지 않는다. `complex`를 사용하는 것이 복소수를 만들 때 효율적이이다. 이렇게 만들면 곱셈과 덧셈 연산자를 사용하지 않는다.

```
julia> a = 1; b = 2; complex(a, b)
1 + 2im
```

특별한 부동 소수점 값들에서 소개한 `Inf`과 `NaN`을 복소수에서도 사용할 수 있다:

```
julia> 1 + Inf*im
1.0 + Inf*im

julia> 1 + NaN*im
1.0 + NaN*im
```

6.2 유리수

Julia는 정수의 비로 유리수를 표현한다. 유리수는 //연산자로 만들 수 있다:

```
julia> 2//3
2//3
```

분모와 분자가 공통 인수를 가지고 있다면, 이들은 자동으로 약분된다:

```
julia> 6//9
2//3

julia> -4//8
-1//2

julia> 5//-15
-1//3

julia> -4//-12
1//3
```

분자와 분모가 서로소인 상태는 유일하며, 두 유리수가 같은지 보려면 각 분자와 분모가 같은지 보면 된다. 유리수의 분자와 분모는 numerator와 denominator함수로 확인할 수 있다:

```
julia> numerator(2//3)
2

julia> denominator(2//3)
3
```

비교연산자는 유리수에 대하여 정의되어 있으므로 분자와 분모를 직접 비교할 일은 적을 것이다:

```
julia> 2//3 == 6//9
true

julia> 2//3 == 9//27
false

julia> 3//7 < 1//2
true

julia> 3//4 > 2//3
true

julia> 2//4 + 1//6
2//3

julia> 5//12 - 1//4
1//6

julia> 5//8 * 3//12
5//32

julia> 6//5 / 10//7
21//25
```

유리수는 쉽게 실수형으로 변환할 수 있다:

```
julia> float(3//4)
0.75
```

유리수를 실수와 비교할 때는 유리수를 실수로 형 변환을 하고 비교하도록 설계되었다(단, `a == 0`이고 `b == 0`인 경우 제외):

```
julia> a = 1; b = 2;

julia> isequal(float(a//b), a/b)
true
```

유리수를 사용하면 무한대를 다음과 같이 정의할 수 있다:

```
julia> 5//0
1//0

julia> -3//0
-1//0

julia> typeof(ans)
Rational{Int64}
```

하지만 유리수에서 NaN를 정의할 수는 없다:

```
julia> 0//0
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
Stacktrace:
[...]
```

유리수는 타입 승격 시스템(promotion system)으로 쉽게 다른 타입의 숫자와 상호작용 할 수 있다:

```
julia> 3//5 + 1
8//5

julia> 3//5 - 0.5
0.09999999999999998

julia> 2//7 * (1 + 2im)
2//7 + 4//7*im

julia> 2//7 * (1.5 + 2im)
0.42857142857142855 + 0.5714285714285714im

julia> 3//2 / (1 + 2im)
3//10 - 3//5*im

julia> 1//2 + 2im
1//2 + 2//1*im

julia> 1 + 2//3im
1//1 - 2//3*im
```

```
julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.0033333333333332993
```

Chapter 7

Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters `A`, `B`, `C`, etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the ASCII standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean. The Unicode standard tackles the complexities of what exactly a character is, and is generally accepted as the definitive standard addressing this problem. Depending on your needs, you can either ignore these complexities entirely and just pretend that only ASCII characters exist, or you can write code that can handle any of the characters or encodings that one may encounter when handling non-ASCII text. Julia makes dealing with plain ASCII text simple and efficient, and handling Unicode is as simple and efficient as possible. In particular, you can write C-style string code to process ASCII strings, and they will work as expected, both in terms of performance and semantics. If such code encounters non-ASCII text, it will gracefully fail with a clear error message, rather than silently introducing corrupt results. When this happens, modifying the code to handle non-ASCII data is straightforward.

There are a few noteworthy high-level features about Julia's strings:

- The built-in concrete type used for strings (and string literals) in Julia is `String`. This supports the full range of Unicode characters via the UTF-8 encoding. (A `transcode` function is provided to convert to/from other Unicode encodings.)
- All string types are subtypes of the abstract type `AbstractString`, and external packages define additional `AbstractString` subtypes (e.g. for other encodings). If you define a function expecting a string argument, you should declare the type as `AbstractString` in order to accept any string type.
- Like C and Java, but unlike most dynamic languages, Julia has a first-class type for representing a single

character, called `AbstractChar`. The built-in `Char` subtype of `AbstractChar` is a 32-bit primitive type that can represent any Unicode character (and which is based on the UTF-8 encoding).

- As in Java, strings are immutable: the value of an `AbstractString` object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.
- Conceptually, a string is a partial function from indices to characters: for some index values, no character value is returned, and instead an exception is thrown. This allows for efficient indexing into strings by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode strings.

7.1 Characters

A `Char` value represents a single character: it is just a 32-bit primitive type with a special literal representation and appropriate arithmetic behaviors, and which can be converted to a numeric value representing a Unicode code point. (Julia packages may define other subtypes of `AbstractChar`, e.g. to optimize operations for other text encodings.) Here is how `Char` values are input and shown:

```
julia> 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> typeof(ans)
Char
```

You can easily convert a `Char` to its integer value, i.e. code point:

```
julia> Int('x')
120

julia> typeof(ans)
Int64
```

On 32-bit architectures, `typeof(ans)` will be `Int32`. You can convert an integer value back to a `Char` just as easily:

```
julia> Char(120)
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

Not all integer values are valid Unicode code points, but for performance, the `Char` conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the `isvalid` function:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category In: Invalid, too high)

julia> isvalid(Char, 0x110000)
false
```

As of this writing, the valid Unicode code points are `U+00` through `U+d7ff` and `U+e000` through `U+10ffff`. These have not all been assigned intelligible meanings yet, nor are they necessarily interpretable by applications, but all of these values are considered to be valid Unicode characters.

You can input any Unicode character in single quotes using `\u` followed by up to four hexadecimal digits or `\U` followed by up to eight hexadecimal digits (the longest valid value only requires six):

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\u2200'
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10ffff (category Cn: Other, not assigned)
```

Julia uses your system's locale and language settings to determine which characters can be printed as-is and which must be output using the generic, escaped `\u` or `\U` input forms. In addition to these Unicode escape forms, all of C's traditional escaped input forms can also be used:

```
julia> Int('\0')
0

julia> Int('\t')
9

julia> Int('\n')
10

julia> Int('\e')
```

```
27

julia> Int('\x7f')
127

julia> Int('\177')
127
```

You can do comparisons and a limited amount of arithmetic with `Char` values:

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'A' <= 'X' <= 'Z'
true

julia> 'x' - 'a'
23

julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

7.2 String Basics

String literals are delimited by double quotes or triple double quotes:

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

If you want to extract a character from a string, you index into it:

```
julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)
```

```
julia> str[6]
',': ASCII/Unicode U+002c (category Po: Punctuation, other)

julia> str[end]
'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Many Julia objects, including strings, can be indexed with integers. The index of the first element (the first character of a string) is returned by `firstindex(str)`, and the index of the last element (character) with `lastindex(str)`. The keyword `end` can be used inside an indexing operation as shorthand for the last index along the given dimension. String indexing, like most indexing in Julia, is 1-based: `firstindex` always returns `1` for any `AbstractString`. As we will see below, however, `lastindex(str)` is not in general the same as `length(str)` for a string, because some Unicode characters can occupy multiple "code units".

You can perform arithmetic and other operations with `end`, just like a normal value:

```
julia> str[end-1]
'.': ASCII/Unicode U+002e (category Po: Punctuation, other)

julia> str[end÷2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Using an index less than 1 or greater than `end` raises an error:

```
julia> str[0]
ERROR: BoundsError: attempt to access String
  at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access String
  at index [15]
[...]
```

You can also extract a substring using range indexing:

```
julia> str[4:9]
"lo, wo"
```

Notice that the expressions `str[k]` and `str[k:k]` do not give the same result:

```
julia> str[6]
',': ASCII/Unicode U+002c (category Po: Punctuation, other)

julia> str[6:6]
","
```

The former is a single character value of type `Char`, while the latter is a string value that happens to contain only a single character. In Julia these are very different things.

Range indexing makes a copy of the selected part of the original string. Alternatively, it is possible to create a view into a string using the type `SubString`, for example:

```
julia> str = "long string"
"long string"

julia> substr = SubString(str, 1, 4)
"long"

julia> typeof(substr)
SubString{String}
```

Several standard functions like `chop`, `chomp` or `strip` return a `SubString`.

7.3 Unicode and UTF-8

Julia fully supports Unicode characters and strings. As discussed above, in character literals, Unicode code points can be represented using Unicode `\u` and `\U` escape sequences, as well as all the standard C escape sequences. These can likewise be used to write string literals:

```
julia> s = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

Whether these Unicode characters are displayed as escapes or shown as special characters depends on your terminal's locale settings and its support for Unicode. String literals are encoded using the UTF-8 encoding. UTF-8 is a variable-width encoding, meaning that not all characters are encoded in the same number of bytes ("code units"). In UTF-8, ASCII characters — i.e. those with code points less than 0x80 (128) – are encoded as they are in ASCII, using a single byte, while code points 0x80 and above are encoded using multiple bytes — up to four per character.

String indices in Julia refer to code units (= bytes for UTF-8), the fixed-width building blocks that are used to encode arbitrary characters (code points). This means that not every index into a `String` is necessarily a valid index for a character. If you index into a string at such an invalid byte index, an error is thrown:

```
julia> s[1]
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> s[2]
ERROR: StringIndexError("∀ x ∃ y", 2)
[...]

julia> s[3]
ERROR: StringIndexError("∀ x ∃ y", 3)
Stacktrace:
[...]

julia> s[4]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

In this case, the character `∀` is a three-byte character, so the indices 2 and 3 are invalid and the next character's index is 4; this next valid index can be computed by `nextind(s,1)`, and the next index after that by `nextind(s,4)` and so on.

Since `end` is always the last valid index into a collection, `end-1` references an invalid byte index if the second-to-last character is multibyte.

```
julia> s[end-1]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)

julia> s[end-2]
ERROR: StringIndexError("∀ x ∃ y", 9)
Stacktrace:
[...]

julia> s[prevind(s, end, 2)]
'∃': Unicode U+2203 (category Sm: Symbol, math)
```

The first case works, because the last character `y` and the space are one-byte characters, whereas `end-2` indexes into the middle of the `∃` multibyte representation. The correct way for this case is using `prevind(s, lastindex(s), 2)` or, if you're using that value to index into `s` you can write `s[prevind(s, end, 2)]` and `end` expands to `lastindex(s)`.

Extraction of a substring using range indexing also expects valid byte indices or an error is thrown:

```
julia> s[1:1]
"∀"

julia> s[1:2]
ERROR: StringIndexError("∀ x ∃ y", 2)
Stacktrace:
[...]

julia> s[1:4]
"∀ "
```

Because of variable-length encodings, the number of characters in a string (given by `length(s)`) is not always the same as the last index. If you iterate through the indices 1 through `lastindex(s)` and index into `s`, the sequence of characters returned when errors aren't thrown is the sequence of characters comprising the string `s`. Thus we have the identity that `length(s) <= lastindex(s)`, since each character in a string must have its own index. The following is an inefficient and verbose way to iterate through the characters of `s`:

```
julia> for i = firstindex(s):lastindex(s)
           try
               println(s[i])
           catch
               # ignore the index error
           end
       end
∀

x

∃

y
```

The blank lines actually have spaces on them. Fortunately, the above awkward idiom is unnecessary for iterating through the characters in a string, since you can just use the string as an iterable object, no exception handling required:

```
julia> for c in s
           println(c)
       end
∀

x

∃

y
```

If you need to obtain valid indices for a string, you can use the `nextind` and `prevind` functions to increment/decrement to the next/previous valid index, as mentioned above. You can also use the `eachindex` function to iterate over the valid character indices:

```
julia> collect(eachindex(s))
7-element Array{Int64,1}:
  1
  4
  5
  6
  7
 10
 11
```

To access the raw code units (bytes for UTF-8) of the encoding, you can use the `codeunit(s,i)` function, where the index `i` runs consecutively from `1` to `ncodeunits(s)`. The `codeunits(s)` function returns an `AbstractVector{UInt8}` wrapper that lets you access these raw codeunits (bytes) as an array.

Strings in Julia can contain invalid UTF-8 code unit sequences. This convention allows to treat any byte sequence as a `String`. In such situations a rule is that when parsing a sequence of code units from left to right characters are formed by the longest sequence of 8-bit code units that matches the start of one of the following bit patterns (each `x` can be `0` or `1`):

- `0xxxxxxx`;
- `110xxxxx 10xxxxxx`;
- `1110xxxx 10xxxxxx 10xxxxxx`;

- 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx;
- 10xxxxxx;
- 11111xxx.

In particular this means that overlong and too-high code unit sequences and prefixes thereof are treated as a single invalid character rather than multiple invalid characters. This rule may be best explained with an example:

```
julia> s = "\xc0\xa0\xe2\x88\xe2¦"
"\xc0\xa0\xe2\x88\xe2¦"

julia> foreach(display, s)
'\xc0\xa0': [overlong] ASCII/Unicode U+0020 (category Zs: Separator, space)
'\xe2\x88': Malformed UTF-8 (category Ma: Malformed, bad data)
'\xe2': Malformed UTF-8 (category Ma: Malformed, bad data)
'¦': ASCII/Unicode U+007c (category Sm: Symbol, math)

julia> isvalid.(collect(s))
4-element BitArray{1}:
 0
 0
 0
 1

julia> s2 = "\xf7\xbf\xbf\xbf"
"\U1fffff"

julia> foreach(display, s2)
'\U1fffff': Unicode U+1fffff (category In: Invalid, too high)
```

We can see that the first two code units in the string `s` form an overlong encoding of space character. It is invalid, but is accepted in a string as a single character. The next two code units form a valid start of a three-byte UTF-8 sequence. However, the fifth code unit `\xe2` is not its valid continuation. Therefore code units 3 and 4 are also interpreted as malformed characters in this string. Similarly code unit 5 forms a malformed character because ¦ is not a valid continuation to it. Finally the string `s2` contains one too high code point.

Julia uses the UTF-8 encoding by default, and support for new encodings can be added by packages. For example, the LegacyStrings.jl package implements `UTF16String` and `UTF32String` types. Additional discussion of other encodings and how to implement support for them is beyond the scope of this document for the time being. For further discussion

of UTF-8 encoding issues, see the section below on byte array literals. The `transcode` function is provided to convert data between the various UTF-xx encodings, primarily for working with external data and libraries.

7.4 Concatenation

One of the most common and useful string operations is concatenation:

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

It's important to be aware of potentially dangerous situations such as concatenation of invalid UTF-8 strings. The resulting string may contain different characters than the input strings, and its number of characters may be lower than sum of numbers of characters of the concatenated strings, e.g.:

```
julia> a, b = "\xe2\x88", "\x80"
("\xe2\x88", "\x80")

julia> c = a*b
"∀"

julia> collect.([a, b, c])
3-element Array{Array{Char,1},1}:
 ['\xe2\x88']
 ['\x80']
 ['∀']

julia> length.([a, b, c])
3-element Array{Int64,1}:
 1
 1
 1
```

This situation can happen only for invalid UTF-8 strings. For valid UTF-8 strings concatenation preserves all characters in strings and additivity of string lengths.

Julia also provides * for string concatenation:

```
julia> greet * ", " * whom * ".\n"
"Hello, world.\n"
```

While * may seem like a surprising choice to users of languages that provide + for string concatenation, this use of * has precedent in mathematics, particularly in abstract algebra.

In mathematics, + usually denotes a commutative operation, where the order of the operands does not matter. An example of this is matrix addition, where `A + B == B + A` for any matrices `A` and `B` that have the same shape. In contrast, * typically denotes a noncommutative operation, where the order of the operands does matter. An example of this is matrix multiplication, where in general `A * B != B * A`. As with matrix multiplication, string concatenation is noncommutative: `greet * whom != whom * greet`. As such, * is a more natural choice for an infix string concatenation operator, consistent with common mathematical use.

More precisely, the set of all finite-length strings S together with the string concatenation operator * forms a free monoid (S, *). The identity element of this set is the empty string, "". Whenever a free monoid is not commutative, the operation is typically represented as `\cdot`, *, or a similar symbol, rather than +, which as stated usually implies commutativity.

7.5 Interpolation

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to `string` or repeated multiplications, Julia allows interpolation into string literals using $, as in Perl:

```
julia> "$greet, $whom.\n"
"Hello, world.\n"
```

This is more readable and convenient and equivalent to the above string concatenation – the system rewrites this apparent single string literal into the call `string(greet, ", ", whom, ".\n")`.

The shortest complete expression after the $ is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

Both concatenation and string interpolation call `string` to convert objects into string form. However, `string` actually just returns the output of `print`, so new types should add methods to `print` or `show` instead of `string`.

Most non-`AbstractString` objects are converted to strings closely corresponding to how they are entered as literal expressions:

```
julia> v = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> "v: $v"
"v: [1, 2, 3]"
```

`string` is the identity for `AbstractString` and `AbstractChar` values, so these are interpolated into strings as themselves, unquoted and unescaped:

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> "hi, $c"
"hi, x"
```

To include a literal `$` in a string literal, escape it with a backslash:

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

7.6 Triple-Quoted String Literals

When strings are created using triple-quotes (`"""..."""`) they have some special behavior that can be useful for creating longer blocks of text.

First, triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented. For example:

```
julia> str = """
           Hello,
           world.
         """
"  Hello,\n  world.\n"
```

In this case the final (empty) line before the closing """ sets the indentation level.

The dedentation level is determined as the longest common starting sequence of spaces or tabs in all lines, excluding the line following the opening """ and lines containing only spaces or tabs (the line containing the closing """ is always included). Then for all lines, excluding the text following the opening """, the common starting sequence is removed (including lines containing only spaces and tabs if they start with this sequence), e.g.:

```
julia> """    This
         is
           a test"""
"    This\nis\n  a test"
```

Next, if the opening """ is followed by a newline, the newline is stripped from the resulting string.

```
"""hello"""
```

is equivalent to

```
"""
hello"""
```

but

```
"""

hello"""
```

will contain a literal newline at the beginning.

Stripping of the newline is performed after the dedentation. For example:

```
julia> """
         Hello,
         world."""
"Hello,\nworld."
```

Trailing whitespace is left unaltered.

Triple-quoted string literals can contain " symbols without escaping.

Note that line breaks in literal strings, whether single- or triple-quoted, result in a newline (LF) character `\n` in the string, even if your editor uses a carriage return `\r` (CR) or CRLF combination to end lines. To include a CR in a string, use an explicit escape `\r`; for example, you can enter the literal string `"a CRLF line ending\r\n"`.

7.7 Common Operations

You can lexicographically compare strings using the standard comparison operators:

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true

julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

You can search for the index of a particular character using the `findfirst` and `findlast` functions:

```
julia> findfirst(isequal('o'), "xylophone")
4

julia> findlast(isequal('o'), "xylophone")
7

julia> findfirst(isequal('z'), "xylophone")
```

You can start the search for a character at a given offset by using the functions `findnext` and `findprev`:

```
julia> findnext(isequal('o'), "xylophone", 1)
4
```

```
julia> findnext(isequal('o'), "xylophone", 5)
7

julia> findprev(isequal('o'), "xylophone", 5)
4

julia> findnext(isequal('o'), "xylophone", 8)
```

You can use the `occursin` function to check if a substring is found within a string:

```
julia> occursin("world", "Hello, world.")
true

julia> occursin("o", "Xylophon")
true

julia> occursin("a", "Xylophon")
false

julia> occursin('o', "Xylophon")
true
```

The last example shows that `occursin` can also look for a character literal.

Two other handy string functions are `repeat` and `join`:

```
julia> repeat(".:Z:.", 10)
".:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:."

julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

Some other useful functions include:

- `firstindex(str)` gives the minimal (byte) index that can be used to index into `str` (always 1 for strings, not necessarily true for other containers).
- `lastindex(str)` gives the maximal (byte) index that can be used to index into `str`.
- `length(str)` the number of characters in `str`.

- `length(str, i, j)` the number of valid character indices in `str` from `i` to `j`.
- `ncodeunits(str)` number of code units in a string.
- `codeunit(str, i)` gives the code unit value in the string `str` at index `i`.
- `thisind(str, i)` given an arbitrary index into a string find the first index of the character into which the index points.
- `nextind(str, i, n=1)` find the start of the `n`th character starting after index `i`.
- `prevind(str, i, n=1)` find the start of the `n`th character starting before index `i`.

7.8 Non-Standard String Literals

There are situations when you want to construct a string or use string semantics, but the behavior of the standard string construct is not quite what is needed. For these kinds of situations, Julia provides non-standard string literals. A non-standard string literal looks like a regular double-quoted string literal, but is immediately prefixed by an identifier, and doesn't behave quite like a normal string literal. Regular expressions, byte array literals and version number literals, as described below, are some examples of non-standard string literals. Other examples are given in the Metaprogramming section.

7.9 Regular Expressions

Julia has Perl-compatible regular expressions (regexes), as provided by the PCRE library (a description of the syntax can be found here). Regular expressions are related to strings in two ways: the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with `r`. The most basic regular expression literal without any options turned on just uses `r"..."`:

```
julia> r"^\s*(?:#¦$)"
r"^\s*(?:#¦$)"

julia> typeof(ans)
Regex
```

To check if a regex matches a string, use `occursin`:

```
julia> occursin(r"^\s*(?:#¦$)", "not a comment")
false
```

```
julia> occursin(r"^\s*(?:#¦$)", "# a comment")
true
```

As one can see here, `occursin` simply returns true or false, indicating whether a match for the given regex occurs in the string. Commonly, however, one wants to know not just whether a string matched, but also how it matched. To capture this information about a match, use the `match` function instead:

```
julia> match(r"^\s*(?:#¦$)", "not a comment")

julia> match(r"^\s*(?:#¦$)", "# a comment")
RegexMatch("#")
```

If the regular expression does not match the given string, `match` returns `nothing` – a special value that does not print anything at the interactive prompt. Other than not printing, it is a completely normal value and you can test for it programmatically:

```
m = match(r"^\s*(?:#¦$)", line)
if m === nothing
    println("not a comment")
else
    println("blank or comment")
end
```

If a regular expression does match, the value returned by `match` is a `RegexMatch` object. These objects record how the expression matches, including the substring that the pattern matches and any captured substrings, if there are any. This example only captures the portion of the substring that matches, but perhaps we want to capture any non-blank text after the comment character. We could do the following:

```
julia> m = match(r"^\s*(?:#\s*(.*?)\s*$¦$)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

When calling `match`, you have the option to specify an index at which to start the search. For example:

```
julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",1)
RegexMatch("1")
```

```
julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",6)
RegexMatch("2")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",11)
RegexMatch("3")
```

You can extract the following info from a `RegexMatch` object:

- the entire substring matched: `m.match`
- the captured substrings as an array of strings: `m.captures`
- the offset at which the whole match begins: `m.offset`
- the offsets of the captured substrings as a vector: `m.offsets`

For when a capture doesn't match, instead of a substring, `m.captures` contains `nothing` in that position, and `m.offsets` has a zero offset (recall that indices in Julia are 1-based, so a zero offset into a string is invalid). Here is a pair of somewhat contrived examples:

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")

julia> m.match
"acd"

julia> m.captures
3-element Array{Union{Nothing, SubString{String}},1}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"

julia> m.captures
3-element Array{Union{Nothing, SubString{String}},1}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 1
 0
 2
```

It is convenient to have captures returned as an array so that one can use destructuring syntax to bind them to local variables:

```
julia> first, second, third = m.captures; first
"a"
```

Captures can also be accessed by indexing the `RegexMatch` object with the number or name of the capture group:

```
julia> m=match(r"(?<hour>\d+):(?<minute>\d+)","12:45")
RegexMatch("12:45", hour="12", minute="45")

julia> m[:minute]
"45"

julia> m[2]
"45"
```

Captures can be referenced in a substitution string when using `replace` by using `\n` to refer to the nth capture group and prefixing the substitution string with `s`. Capture group 0 refers to the entire match object. Named capture groups can be referenced in the substitution with `\g<groupname>`. For example:

```
julia> replace("first second", r"(\w+) (?<agroup>\w+)" => s"\g<agroup> \1")
"second first"
```

Numbered capture groups can also be referenced as `\g<n>` for disambiguation, as in:

```
julia> replace("a", r"." => s"\g<0>1")
"a1"
```

You can modify the behavior of regular expressions by some combination of the flags `i`, `m`, `s`, and `x` after the closing double quote mark. These flags have the same meaning as they do in Perl, as explained in this excerpt from the perlre manpage:

```
i   Do case-insensitive pattern matching.

    If locale matching rules are in effect, the case map is taken
    from the current locale for code points less than 255, and
    from Unicode rules for larger code points. However, matches
    that would cross the Unicode rules/non-Unicode rules boundary
    (ords 255/256) will not succeed.

m   Treat string as multiple lines.  That is, change "^" and "$"
    from matching the start or end of the string to matching the
    start or end of any line anywhere within the string.

s   Treat string as single line.  That is, change "." to match any
    character whatsoever, even a newline, which normally it would
    not match.

    Used together, as r""ms, they let the "." match any character
    whatsoever, while still allowing "^" and "$" to match,
    respectively, just after and just before newlines within the
    string.

x   Tells the regular expression parser to ignore most whitespace
    that is neither backslashed nor within a character class. You
```

```
    can use this to break up your regular expression into
    (slightly) more readable parts. The '#' character is also
    treated as a metacharacter introducing a comment, just as in
    ordinary code.
```

For example, the following regex has all three flags turned on:

```
julia> r"a+.*b+.*?d$"ism
r"a+.*b+.*?d$"ims

julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

The `r"..."` literal is constructed without interpolation and unescaping (except for quotation mark " which still has to be escaped). Here is an example showing the difference from standard string literals:

```
julia> x = 10
10

julia> r"$x"
r"$x"

julia> "$x"
"10"

julia> r"\x"
r"\x"

julia> "\x"
ERROR: syntax: invalid escape sequence
```

Triple-quoted regex strings, of the form `r"""..."""`, are also supported (and may be convenient for regular expressions containing quotation marks or newlines).

The `Regex()` constructor may be used to create a valid regex string programmatically. This permits using the contents of string variables and other string operations when constructing the regex string. Any of the regex codes above can be used within the single string argument to `Regex()`. Here are some examples:

```
julia> using Dates
```

```
julia> d = Date(1962,7,10)
1962-07-10

julia> regex_d = Regex("Day " * string(day(d)))
r"Day 10"

julia> match(regex_d, "It happened on Day 10")
RegexMatch("Day 10")

julia> name = "Jon"
"Jon"

julia> regex_name = Regex("[\"( ]$name[\") ]")  # interpolate value of name
r"[\"( ]Jon[\") ]"

julia> match(regex_name," Jon ")
RegexMatch(" Jon ")

julia> match(regex_name,"[Jon]") === nothing
true
```

7.10 Byte Array Literals

Another useful non-standard string literal is the byte-array string literal: `b"..."`. This form lets you use string notation to express read only literal byte arrays – i.e. arrays of `UInt8` values. The type of those objects is `CodeUnits{UInt8, String}`. The rules for byte array literals are the following:

- ASCII characters and ASCII escapes produce a single byte.
- `\x` and octal escape sequences produce the byte corresponding to the escape value.
- Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

There is some overlap between these rules since the behavior of `\x` and octal escapes less than 0x80 (128) are covered by both of the first two rules, but here these rules agree. Together, these rules allow one to easily use ASCII characters, arbitrary byte values, and UTF-8 sequences to produce arrays of bytes. Here is an example using all three:

```
julia> b"DATA\xff\u2200"
8-element Base.CodeUnits{UInt8,String}:
 0x44
```

```
 0x41
 0x54
 0x41
 0xff
 0xe2
 0x88
 0x80
```

The ASCII string "DATA" corresponds to the bytes 68, 65, 84, 65. `\xff` produces the single byte 255. The Unicode escape `\u2200` is encoded in UTF-8 as the three bytes 226, 136, 128. Note that the resulting byte array does not correspond to a valid UTF-8 string:

```
julia> isvalid("DATA\xff\u2200")
false
```

As it was mentioned `CodeUnits{UInt8,String}` type behaves like read only array of `UInt8` and if you need a standard vector you can convert it using `Vector{UInt8}`:

```
julia> x = b"123"
3-element Base.CodeUnits{UInt8,String}:
 0x31
 0x32
 0x33

julia> x[1]
0x31

julia> x[1] = 0x32
ERROR: setindex! not defined for Base.CodeUnits{UInt8,String}
[...]

julia> Vector{UInt8}(x)
3-element Array{UInt8,1}:
 0x31
 0x32
 0x33
```

Also observe the significant distinction between `\xff` and `\uff`: the former escape sequence encodes the byte 255, whereas the latter escape sequence represents the code point 255, which is encoded as two bytes in UTF-8:

```
julia> b"\xff"
1-element Base.CodeUnits{UInt8,String}:
 0xff

julia> b"\uff"
2-element Base.CodeUnits{UInt8,String}:
 0xc3
 0xbf
```

Character literals use the same behavior.

For code points less than `\u80`, it happens that the UTF-8 encoding of each code point is just the single byte produced by the corresponding `\x` escape, so the distinction can safely be ignored. For the escapes `\x80` through `\xff` as compared to `\u80` through `\uff`, however, there is a major difference: the former escapes all encode single bytes, which – unless followed by very specific continuation bytes – do not form valid UTF-8 data, whereas the latter escapes all represent Unicode code points with two-byte encodings.

If this is all extremely confusing, try reading "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets". It's an excellent introduction to Unicode and UTF-8, and may help alleviate some confusion regarding the matter.

7.11 Version Number Literals

Version numbers can easily be expressed with non-standard string literals of the form `v"..."`. Version number literals create `VersionNumber` objects which follow the specifications of semantic versioning, and therefore are composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. For example, `v"0.2.1-rc1+win64"` is broken into major version `0`, minor version `2`, patch version `1`, pre-release `rc1` and build `win64`. When entering a version literal, everything except the major version number is optional, therefore e.g. `v"0.2"` is equivalent to `v"0.2.0"` (with empty pre-release/build annotations), `v"2"` is equivalent to `v"2.0.0"`, and so on.

`VersionNumber` objects are mostly useful to easily and correctly compare two (or more) versions. For example, the constant `VERSION` holds Julia version number as a `VersionNumber` object, and therefore one can define some version-specific behavior using simple statements as:

```
if v"0.2" <= VERSION < v"0.3-"
    # do something specific to 0.2 release series
end
```

Note that in the above example the non-standard version number `v"0.3-"` is used, with a trailing `-`: this notation is a Julia extension of the standard, and it's used to indicate a version which is lower than any `0.3` release, including

all of its pre-releases. So in the above example the code would only run with stable `0.2` versions, and exclude such versions as `v"0.3.0-rc1"`. In order to also allow for unstable (i.e. pre-release) `0.2` versions, the lower bound check should be modified like this: `v"0.2-" <= VERSION`.

Another non-standard version specification extension allows one to use a trailing `+` to express an upper limit on build versions, e.g. `VERSION > v"0.2-rc1+"` can be used to mean any version above `0.2-rc1` and any of its builds: it will return `false` for version `v"0.2-rc1+win64"` and `true` for `v"0.2-rc2"`.

It is good practice to use such special versions in comparisons (particularly, the trailing `-` should always be used on upper bounds unless there's a good reason not to), but they must not be used as the actual version number of anything, as they are invalid in the semantic versioning scheme.

Besides being used for the `VERSION` constant, `VersionNumber` objects are widely used in the `Pkg` module, to specify packages versions and their dependencies.

7.12 Raw String Literals

Raw strings without interpolation or unescaping can be expressed with non-standard string literals of the form `raw"..."`. Raw string literals create ordinary `String` objects which contain the enclosed contents exactly as entered with no interpolation or unescaping. This is useful for strings which contain code or markup in other languages which use $ or \ as special characters.

The exception is that quotation marks still must be escaped, e.g. `raw"\""` is equivalent to `"\""`. To make it possible to express all strings, backslashes then also must be escaped, but only when appearing right before a quote character:

```
julia> println(raw"\\ \\\"")
\\ \"
```

Notice that the first two backslashes appear verbatim in the output, since they do not precede a quote character. However, the next backslash character escapes the backslash that follows it, and the last backslash escapes a quote, since these backslashes appear before a quote.

Chapter 8

함수

함수는 인자를 받아 값을 반환하는 객체이다. Julia에서 정의하는 함수는 실행 상황에 영향을 받는다는 점에서 수학적 정의에 따른 함수와는 조금 다르다. 아래는 Julia에서 함수를 정의하는 가장 기본적인 방법이다:

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

아래와 같이 함수를 정의하는 방법도 있다:

```
julia> f(x,y) = x + y
f (generic function with 1 method)
```

위처럼 "할당 형식(assignment form)"으로 선언할 경우 복합 표현이더라도 한 줄로 표현해야 한다(복합 표현을 자세하고 알고 싶다면?). 이렇게 함수를 표현하는 경우는 Julia에 흔한 일이고, 때론 코드 가독성을 높여준다.

다른 언어처럼 소괄호를 통해 함수 인자를 전달한다:

```
julia> f(2,3)
5
```

소괄호가 없는 `f`는 함수 객체로써 하나의 값으로 취급할 수 있다:

```
julia> g = f;

julia> g(2,3)
5
```

함수의 이름은 유니코드라면 무엇이든지 가능하다:

```
julia> ∑(x,y) = x + y
∑ (generic function with 1 method)

julia> ∑(2, 3)
5
```

8.1 인자 전달 방식

함수에 인자를 줄 때 Julia는 "공유를 통한 전달(pass-by-sharing)"을 한다. 이 말인즉슨, 객체를 복사하지 않고 공유한다는 뜻이다. 전달된 인자는 함수 안에 있는 변수에 할당되고, 함수 안의 변수는 단지 그 객체를 가리킬 뿐이다. `Array`와 같은 mutable 객체가 함수 안에서 변하면, 함수 밖에서도 그 변화를 볼 수 있다. 이런 방식은 Scheme, Python, Ruby, Perl 그리고 대부분의 Lisp와 같은 동적언어가 채택한 방식이다.

8.2 return 키워드

함수가 반환하는 값은 암묵적으로 가장 마지막으로 계산된 값이다. 이전의 예제 함수 `f`에서는 `x+y`의 값이 반환될 것이다. 다른 프로그래밍 언어처럼 `return`과 반환값이 명시적으로 선언될 경우, 함수는 즉시 종료되고 `return` 앞에 있는 식을 계산하고 반환할 것이다:

```
function g(x,y)
    return x * y
    x + y
end
```

직접 테스트해보자:

```
julia> f(x,y) = x + y
f (generic function with 1 method)

julia> function g(x,y)
           return x * y
           x + y
       end
g (generic function with 1 method)

julia> f(2,3)
5
```

```
julia> g(2,3)
6
```

함수 g에서 x+y는 절대 실행되지 않기 때문에, 이 부분을 빼고 x*y만 남겨놔도 똑같이 작동한다. return을 직접 선언하는 방식은 조건문과 같이 코드의 흐름을 바꾸는 구문과 사용했을 때 빛을 발한다. 아래에 직각 삼각형에서 밑변 x와 높이 y가 주어졌을 때 빗변의 길이는 구하는 예제로 확인할 수 있다. 아래 함수는 overflow를 없애기 위해 조건문을 사용했다:

```
julia> function hypot(x,y)
           x = abs(x)
           y = abs(y)
           if x > y
               r = y/x
               return x*sqrt(1+r*r)
           end
           if y == 0
               return zero(x)
           end
           r = x/y
           return y*sqrt(1+r*r)
       end
hypot (generic function with 1 method)

julia> hypot(3, 4)
5.0
```

위 함수는 경우에 따라 세 가지 방법으로 값을 반환한다. 마지막에 return은 생략해도 된다.

8.3 반환 타입

반환값의 타입은 ::로 명시할 수 있으며, 이 경우 반환값이 자동 형변환된다.

```
julia> function g(x, y)::Int8
           return x * y
       end;

julia> typeof(g(1, 2))
Int8
```

위 함수는 x와 y의 타입에 상관없이 반환값은 Int8로 정해져있다. 타입에 대해 자세히 알고 싶다면 Type Declarations을 참고하자.

8.4 반환값이 없는 함수

함수가 값을 반환할 필요가 없을 경우, Julia 언어 내에서는 관습적으로 nothing을 반환한다:

```
function printx(x)
    println("x = $x")
    return nothing
end
```

This is a convention in the sense that `nothing` is not a Julia keyword but a only singleton object of type `Nothing`. Also, you may notice that the `printx` function example above is contrived, because `println` already returns `nothing`, so that the `return` line is redundant.

There are two possible shortened forms for the `return nothing` expression. On the one hand, the `return` keyword implicitly returns `nothing`, so it can be used alone. On the other hand, since functions implicitly return their last expression evaluated, `nothing` can be used alone when it's the last expression. The preference for the expression `return nothing` as opposed to `return` or `nothing` alone is a matter of coding style.

8.5 연산자는 함수다

Julia에서 연산자는 특별한 문법을 가진 함수일 뿐이다(&&와 ||는 예외다. 이들은 단락 계산에서 나왔다시피 연산자가 피연산자보다 먼저 계산되기 때문이다). 따라서 연산자는 일반 함수처럼 소괄호를 이용해 인자를 전달할 수 있다:

```
julia> 1 + 2 + 3
6

julia> +(1,2,3)
6
```

infix 표기법(1+2+3)과 함수 표기법은 같은 결과를 낸다. 실제로 Julia는 내부에서 infix 표기를 함수 표기로 바꿔서 계산하기 때문에 같을 수밖에 없다. 연산자가 함수이기 때문에 다음과 같이 사용할 수도 있다:

```
julia> f = +;

julia> f(1,2,3)
6
```

다만 위처럼 함수 이름이 바뀌면 infix 표기법을 사용할 수 없다.

8.6 특별한 이름을 가진 함수

특정 함수는 호출 대신 특수한 문법으로 대체할 수 있다. 그러한 함수는 다음과 같습니다:

문법	함수 이름
`[A B C ...]`	`hcat`
`[A; B; C; ...]`	`vcat`
`[A B; C D; ...]`	`hvcat`
`A'`	`adjoint`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`
`A.n`	`getproperty`
`A.n = x`	`setproperty!`

8.7 익명 함수

Julia에서 함수는 일급 객체다: 변수에 값으로 저장될 수 있고, 해당 변수를 함수로 사용할 수 있다. 또 함수 객체는 다른 함수의 인자가 될 수도 있고 반환값이 될 수도 있다. 함수의 이름이 없어도 함수를 다음과 같은 방법으로 정의할 수 있다:

```
julia> x -> x^2 + 2x - 1
#1 (generic function with 1 method)

julia> function (x)
           x^2 + 2x - 1
       end
#3 (generic function with 1 method)
```

두 방법 모두 x를 받아 `x^2 + 2x - 1`를 반환하는 함수를 만든다. 위와 같은 방식으로 함수를 만들면 함수 이름 대신 컴파일러가 #1, #3과 같은 숫자로 함수를 구분하는 걸 볼 수 있다.

익명 함수는 함수를 함수 인자로 주면서, 한 번 밖에 사용하지 않을 때 유용하다. `map`이 그 중 하나로, 배열이 값 각각을 인자로 받는 함수를 받아 반환값으로 새로운 배열을 만든다:

```
julia> map(round, [1.2,3.5,1.7])
3-element Array{Float64,1}:
```

```
1.0
4.0
2.0
```

위에서는 이미 원하는 함수가 정의되어 있었기 때문에 문제가 없었다. 하지만 그런 함수가 없을 때, 익명 함수를 사용하면 편리하다:

```
julia> map(x -> x^2 + 2x - 1, [1,3,-1])
3-element Array{Int64,1}:
  2
 14
 -2
```

익명 함수에 다중 인자를 사용하려면 `(x,y,z)->2x+y-z`처럼 쓰면 된다. `()->3`처럼 인자를 받지 않는 함수를 정의할 수도 있다. 처음 프로그래밍을 접하면 "인자를 받지 않는 함수를 왜쓰지?"라고 생각할 수 있지만 코딩을 하다보면 여러모로 유용하다.

8.8 튜플

줄리아의 튜플은 함수의 입출력에 중요하게 관여한다. 튜플은 어떤 값이든 저장할 수 있는 고정 크기의 컨테이너이며, 생성 후에는 수정이 불가능(immutable)하다. 튜플은 반점과 소괄호를 이용해 만들고 인덱싱을 통해 값에 접근한다:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"
```

크기가 1인 튜플을 만들고 싶어도 `(1,)`처럼 꼭 반점을 넣어야 한다. `(1)`은 값을 소괄호로 감싼 것으로 취급된다. `()`은 비어 있는 튜플을 생성한다.

8.9 지명 튜플(Named tuple)

튜플의 인자에 이름을 부여할 수 있으며 이를 지명 튜플이라고 한다:

```
julia> x = (a=1, b=1+1)
(a = 1, b = 2)

julia> x.a
1
```

지명 튜플은 이름이 있다는 것을 제외하면 일반적인 튜플과 유사하며, dot 문법을 통해 값에 접근할 수 있다 (`x.a`).

8.10 다중 반환

여러 값을 반환하기 위해 함수는 튜플을 반환한다. 하지만 튜플은 괄호 없이 생성되기도 하고 분리되기도 하므로 명시적으로 튜플을 사용한다는 것을 나타낼 필요가 없다. 이는 우리가 값을 여러 개 반환한다는 환상을 심어준다. 예제로 두 개의 값을 반환하는 상황을 보자:

```
julia> function foo(a,b)
           a+b, a*b
       end
foo (generic function with 1 method)
```

대화형 실행환경에서 함수를 실행하면 튜플이 반환되는 것을 확인할 수 있다:

```
julia> foo(2,3)
(5, 6)
```

보통의 경우 튜플의 값을 변수로 각각 분리하고 사용하기 때문에, Julia는 튜플을 분리할 수 있는 간단한 방법을 제공하여 편의성을 높였다:

```
julia> x, y = foo(2,3)
(5, 6)

julia> x
5

julia> y
6
```

return으로도 다중 변수 반환을 할 수 있다. 아래 예제는 이전 예제와 똑같이 작동한다:

```
function foo(a,b)
    return a+b, a*b
end
```

8.11 인자 분리

The destructuring feature can also be used within a function argument. If a function argument name is written as a tuple (e.g. `(x, y)`) instead of just a symbol, then an assignment `(x, y) = argument` will be inserted for you:

```
julia> minmax(x, y) = (y < x) ? (y, x) : (x, y)

julia> range((min, max)) = max - min

julia> range(minmax(10, 2))
8
```

Notice the extra set of parentheses in the definition of `range`. Without those, `range` would be a two-argument function, and this example would not work.

8.12 가변인자 함수

경우에 따라 함수에 원하는 만큼 인자를 주는 것이 유용할 때도 있다. 이러한 가변인자 함수를 만들려면 함수 인자 선언의 마지막에 `(인자 이름)...`을 넣으면 된다:

```
julia> bar(a,b,x...) = (a,b,x)
bar (generic function with 1 method)
```

위 예제에서 처음 두번째 인자까지는 a와 b에 할당되고, 변수 x에는 나머지 인자들이 튜플로 묶여서 전달된다:

```
julia> bar(1,2)
(1, 2, ())

julia> bar(1,2,3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
```

```
(1, 2, (3, 4))

julia> bar(1,2,3,4,5,6)
(1, 2, (3, 4, 5, 6))
```

가변인자의 개수를 제한하는 방법은 매개변수적으로 제한된 Varargs 메서드에서 확인할 수 있다.

...을 다르게도 활용할 수 있다. interable 객체에 저장된 값 하나하나를 전부 함수 인자로 주고 싶을 때, 해당 변수에 ...을 붙여주면 순서대로 인자를 넣어준다. 아래의 경우 튜플이 알아서 쪼개져 각 인자에 순서대로 들어간다:

```
julia> x = (3, 4)
(3, 4)

julia> bar(1,2,x...)
(1, 2, (3, 4))

julia> x = (2, 3, 4)
(2, 3, 4)

julia> bar(1,x...)
(1, 2, (3, 4))

julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> bar(x...)
(1, 2, (3, 4))
```

물론 interable 객체이기만 하면 위 방법을 사용할 수 있다:

```
julia> x = [3,4]
2-element Array{Int64,1}:
 3
 4

julia> bar(1,2,x...)
(1, 2, (3, 4))

julia> x = [1,2,3,4]
```

```
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> bar(x...)
(1, 2, (3, 4))
```

이 방법은 가변인자 함수가 아니어도 사용할 수 있다:

```
julia> baz(a,b) = a + b;

julia> args = [1,2]
2-element Array{Int64,1}:
 1
 2

julia> baz(args...)
3

julia> args = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)
Closest candidates are:
  baz(::Any, ::Any) at none:1
```

보다시피 인자의 개수가 잘못되면 함수 호출은 실패하고 위와 같은 에러를 보게 될 것이다.

8.13 기본값이 제공된 인자(optional arguments)

기본값이 지정된 함수는 해당 인자를 주지 않아도 잘 작동한다. 예를 들어Dates의 Date타입에 지정된 `Date(y, [m, d])` 함수는 y만 지정하면 m과 d는 1로 자동 지정된다:

```
function Date(y::Int64, m::Int64=1, d::Int64=1)
    err = validargs(Date, y, m, d)
    err === nothing || throw(err)
    return Date(UTD(totaldays(y, m, d)))
end
```

이 예제에 부연설명을 하면, `Date`함수는 `UTInstant{Day}`라는 인자를 받는 다른 매서드 함수 `Date`를 호출한다. 위 함수의 정의에 따라 이 함수에는 인자를 하나, 둘, 혹은 세개를 줄 수 있으며, 인자가 직접 주어지지 않을 경우 `1`이 자동으로 부여됨을 알 수 있다:

```
julia> using Dates

julia> Date(2000, 12, 12)
2000-12-12

julia> Date(2000, 12)
2000-12-01

julia> Date(2000)
2000-01-01
```

기본값 제공은 다중인자 함수의 사용 편의성을 위한 것이다(Note on Optional and keyword Arguments를 보자). 위 예제에서 메서드 함수를 호출한 것을 보면 알 수 있다.

8.14 Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function `plot` that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like `plot(x, y, width=2)`, where we have chosen to specify only line width. Notice that this serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon in the signature:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

When the function is called, the semicolon is optional: one can either call `plot(x, y, width=2)` or `plot(x, y; width=2)`, but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.

Keyword argument default values are evaluated only when necessary (when a corresponding keyword argument is not passed), and in left-to-right order. Therefore default expressions may refer to prior keyword arguments.

The types of keyword arguments can be made explicit as follows:

```
function f(;x::Int=1)
    ###
end
```

Extra keyword arguments can be collected using `...`, as in varargs functions:

```
function f(x; y=0, kwargs...)
    ###
end
```

Inside `f`, `kwargs` will be a key-value iterator over a named tuple. Named tuples (as well as dictionaries with keys of `Symbol`) can be passed as keyword arguments using a semicolon in a call, e.g. `f(x, z=1; kwargs...)`.

If a keyword argument is not assigned a default value in the method definition, then it is required: an `UndefKeywordError` exception will be thrown if the caller does not assign it a value:

```
function f(x; y)
    ###
end
f(3, y=5) # ok, y is assigned
f(3)      # throws UndefKeywordError(:y)
```

One can also pass `key => value` expressions after a semicolon. For example, `plot(x, y; :width => 2)` is equivalent to `plot(x, y, width=2)`. This is useful in situations where the keyword name is computed at runtime.

The nature of keyword arguments makes it possible to specify the same argument more than once. For example, in the call `plot(x, y; options..., width=2)` it is possible that the `options` structure also contains a value for `width`. In

such a case the rightmost occurrence takes precedence; in this example, `width` is certain to have the value `2`. However, explicitly specifying the same keyword argument multiple times, for example `plot(x, y, width=2, width=3)`, is not allowed and results in a syntax error.

8.15 Evaluation Scope of Default Values

When optional and keyword argument default expressions are evaluated, only previous arguments are in scope. For example, given this definition:

```
function f(x, a=b, b=1)
    ###
end
```

the `b` in `a=b` refers to a `b` in an outer scope, not the subsequent argument `b`.

8.16 Do-Block Syntax for Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling `map` on a function with several cases:

```
map(x->begin
          if x < 0 && iseven(x)
              return 0
          elseif x == 0
              return 1
          else
              return x
          end
      end,
    [A, B, C])
```

Julia provides a reserved word `do` for rewriting this code more clearly:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
```

```
        return 1
    else
        return x
    end
end
```

The `do x` syntax creates an anonymous function with argument `x` and passes it as the first argument to `map`. Similarly, `do a,b` would create a two-argument anonymous function, and a plain `do` would declare that what follows is an anonymous function of the form `() -> ...`.

How these arguments are initialized depends on the "outer" function; here, `map` will sequentially set `x` to `A`, `B`, `C`, calling the anonymous function on each, just as would happen in the syntax `map(func, [A, B, C])`.

This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from `map`, such as managing system state. For example, there is a version of `open` that runs code ensuring that the opened file is eventually closed:

```
open("outfile", "w") do io
    write(io, data)
end
```

This is accomplished by the following definition:

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
        close(io)
    end
end
```

Here, `open` first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the `do ... end` block. After your function exits, `open` will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception. (The `try/finally` construct will be described in 제어 흐름.)

With the `do` block syntax, it helps to check the documentation or implementation to know how the arguments of the user function are initialized.

A `do` block, like any other inner function, can "capture" variables from its enclosing scope. For example, the variable `data` in the above example of `open...do` is captured from the outer scope. Captured variables can create performance challenges as discussed in performance tips.

8.17 Function composition and piping

Functions in Julia can be combined by composing or piping (chaining) them together.

Function composition is when you combine functions together and apply the resulting composition to arguments. You use the function composition operator (`∘`) to compose the functions, so `(f ∘ g)(args...)` is the same as `f(g(args...))`.

You can type the composition operator at the REPL and suitably-configured editors using `\circ<tab>`.

For example, the `sqrt` and `+` functions can be composed like this:

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

This adds the numbers first, then finds the square root of the result.

The next example composes three functions and maps the result over an array of strings:

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like this"))
6-element Array{Char,1}:
 'U'
 'N'
 'E'
 'S'
 'E'
 'S'
```

Function chaining (sometimes called "piping" or "using a pipe" to send data to a subsequent function) is when you apply a function to the previous function's output:

```
julia> 1:10 |> sum |> sqrt
7.416198487095663
```

Here, the total produced by `sum` is passed to the `sqrt` function. The equivalent composition would be:

```
julia> (sqrt ∘ sum)(1:10)
7.416198487095663
```

The pipe operator can also be used with broadcasting, as `.|>`, to provide a useful combination of the chaining/piping and dot vectorization syntax (described next).

```
julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Array{Any,1}:
  "A"
  "tsil"
  "Of"
 7
```

8.18 배열에서 사용하는 Dot 문법

수치 계산용 언어에서는 함수의 스칼라 버전이 존재하면 벡터 버전이 자동 지원되는 것은 흔하다. 즉 `f(x)`가 있으면 이를 행렬의 모든 원소에 적용하는 `f(A)`가 지원되기 마련이다. 이런 문법은 데이터 처리를 편리하게 하지만, 몇몇 언어는 성능면에서 문제를 겪어 사용자가 직접 저급 언어의 라이브러리를 사용해 벡터 버전의 함수를 만들기도 한다. Julia는 성능 향상을 위해 이런 노력을 할 필요가 없다. 모든 Julia 함수 `f`는 `f.(A)`이란 문법을 사용해 원소별 연산이 가능하다. 예를 들어 `sin`로 벡터 `A`를 쉽게 계산할 수 있다:

```
julia> A = [1.0, 2.0, 3.0]
3-element Array{Float64,1}:
 1.0
 2.0
 3.0

julia> sin.(A)
3-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

물론 사용자가 `f(A::AbstractArray) = map(f, A)`와 같이 직접 벡터 함수를 만드는 것도 가능하고 `f.(A)`만큼 효율적이다.

More generally, `f.(args...)` is actually equivalent to `broadcast(f, args...)`, which allows you to operate on multiple arrays (even of different shapes), or a mix of arrays and scalars (see Broadcasting). For example, if you have `f(x,y) = 3x + 4y`, then `f.(pi,A)` will return a new array consisting of `f(pi,a)` for each `a` in `A`, and `f.(vector1,vector2)` will return a new vector consisting of `f(vector1[i],vector2[i])` for each index `i` (throwing an exception if the vectors have different length).

```
julia> f(x,y) = 3x + 4y;
```

```
julia> A = [1.0, 2.0, 3.0];

julia> B = [4.0, 5.0, 6.0];

julia> f.(pi, A)
3-element Array{Float64,1}:
 13.42477796076938
 17.42477796076938
 21.42477796076938

julia> f.(A, B)
3-element Array{Float64,1}:
 19.0
 26.0
 33.0
```

Moreover, nested `f.(args...)` calls are fused into a single `broadcast` loop. For example, `sin.(cos.(X))` is equivalent to `broadcast(x -> sin(cos(x)), X)`, similar to `[sin(cos(x)) for x in X]`: there is only a single loop over `X`, and a single array is allocated for the result. [In contrast, `sin(cos(X))` in a typical "vectorized" language would first allocate one temporary array for `tmp=cos(X)`, and then compute `sin(tmp)` in a separate loop, allocating a second array.] This loop fusion is not a compiler optimization that may or may not occur, it is a syntactic guarantee whenever nested `f.(args...)` calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in `sin.(sort(cos.(X)))` the `sin` and `cos` loops cannot be merged because of the intervening `sort` function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is pre-allocated, so that repeated calls do not allocate new arrays over and over again for the results (see Pre-allocating outputs). A convenient syntax for this is `X .= ...`, which is equivalent to `broadcast!(identity, X, ...)` except that, as above, the `broadcast!` loop is fused with any nested "dot" calls. For example, `X .= sin.(Y)` is equivalent to `broadcast!(sin, X, Y)`, overwriting `X` with `sin.(Y)` in-place. If the left-hand side is an array-indexing expression, e.g. `X[2:end] .= sin.(Y)`, then it translates to `broadcast!` on a `view`, e.g. `broadcast!(sin, view(X, 2:lastindex(X)), Y)`, so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can be tedious and lead to code that is difficult to read, the macro `@.` is provided to convert every function call, operation, and assignment in an expression into the "dotted" version.

```
julia> Y = [1.0, 2.0, 3.0, 4.0];

julia> X = similar(Y); # pre-allocate output array
```

```
julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))
4-element Array{Float64,1}:
  0.5143952585235492
 -0.4042391538522658
 -0.8360218615377305
 -0.6080830096407656
```

Binary (or unary) operators like `.+` are handled with the same mechanism: they are equivalent to `broadcast` calls and are fused with other nested "dot" calls. `X .+= Y` etcetera is equivalent to `X .= X .+ Y` and results in a fused in-place assignment; see also dot operators.

You can also combine dot operations with function chaining using `¦>`, as in this example:

```
julia> [1:5;] .¦> [x->x^2, inv, x->2*x, -, isodd]
5-element Array{Real,1}:
    1
    0.5
    6
   -4
 true
```

8.19 Further Reading

We should mention here that this is far from a complete picture of defining functions. Julia has a sophisticated type system and allows multiple dispatch on argument types. None of the examples given here provide any type annotations on their arguments, meaning that they are applicable to all types of arguments. The type system is described in Types and defining a function in terms of methods chosen by multiple dispatch on run-time argument types is described in Methods.

Chapter 9

제어 흐름

Julia는 다양한 제어 흐름 문법을 제공합니다.

- 복합 표현: `begin`및 `(;)`.
- 조건부 계산: `if-elseif-else`및 `?:` (삼항 연산자).
- 단락 계산: `&&`, `||`및 연속 비교문.
- 반복 계산: `while`및 `for`.
- 예외 처리: `try-catch`, `error`및 `throw`.
- 태스크(일명 코루틴): `yieldto`.

태스크를 제외한 나머지 문법들은 프로그래밍 언어의 표준 문법이라 할 수 있습니다. 태스크는 이들과 다르며, 여러 계산을 일시적으로 중단하고 다시 실행하는 능력을 가집니다. 태스크는 강력한 구조이며, Julia는 예외 처리 및 협력적 멀티태스킹을 태스크로 구현합니다. 많은 경우 태스크를 사용할 필요가 없지만, 몇몇 문제는 태스크를 사용함으로써 더 쉽게 해결될 수 있습니다.

9.1 복합 표현

때로는 여러 하위식을 순서대로 계산하는 단 하나의 식이 더 편리하며, 이 경우 마지막 하위식의 값을 그 값으로 반환하게 됩니다. 이를 수행하는 두 개의 Julia 구조가 있습니다: 바로 `begin` 구문과 `(;)` 체인 구문입니다. 두 복합 표현의 반환값은 가장 마지막에 계산된 값입니다. 다음은 `begin` 구문의 예제입니다.

```
julia> z = begin
           x = 1
           y = 2
```

```
            x + y
        end
3
```

위와 같이 식의 길이가 매우 짧고 단순하다면, `(;)` 체인 구문을 사용해 한 줄로 쉽게 표현할 수 있습니다.

```
julia> z = (x = 1; y = 2; x + y)
3
```

이 구문은 함수 문서에 소개된 간결한 단일 행 함수를 정의할 때 특히 유용합니다. 전형적인 구문처럼 보이겠지만, `begin` 블록 내부가 여러 줄일 필요도 없고, `(;)` 체인이 전부 한 줄에서 이루어질 필요도 없습니다.

```
julia> begin x = 1; y = 2; x + y end
3


julia> (x = 1;
        y = 2;
        x + y)
3
```

9.2 조건부 계산

조건부 계산는 논리식의 값에 따라 일부 코드의 실행 여부를 결정합니다. 다음은 `if-elseif-else` 조건 구문의 구조입니다.

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

조건식 `x < y`가 `true`이면 해당 블록이 실행됩니다. 참이 아니라면, 조건식 `x > y`를 계산하고 `true`이면 해당 블록이 실행됩니다. 만약 두 표현 둘 다 참이 아니라면, `else` 블록이 실행됩니다. 다음은 실행 예제입니다.

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
```

```
            elseif x > y
                println("x is greater than y")
            else
                println("x is equal to y")
            end
        end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

elseif와 else 블록은 선택 사항이며, 원하는 만큼 많은 elseif 블록을 사용할 수 있습니다. if-elseif-else 구문 안의 조건식은 어느 한 식이 처음으로 true로 계산될 때까지 계산되고, 그 후에 관련 블록이 실행되며, 이후로는 어떤 식이나 블록도 실행되지 않습니다.

if 블록은 지역 스코프를 만들지 않기 때문에 if 절 안에서 정의된 새로운 변수를 if 블록 밖에서도 사용할 수 있습니다. 따라서, 위에서 정의한 test 함수를 다음과 같이 정의할 수도 있습니다.

```
julia> function test(x,y)
            if x < y
                relation = "less than"
            elseif x == y
                relation = "equal to"
            else
                relation = "greater than"
            end
            println("x is ", relation, " y.")
        end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

relation 변수는 if 블록 안에서 선언되었지만, 블록 밖에서 사용되고 있습니다. 그러나, 모든 코드 경로가 이 구문을 통해 변수 값을 정의할 수 있는지 확인해야 합니다. 위 함수를 다음과 같이 변경하면 런타임 오류가 발생합니다.

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(1,2)
x is less than y.

julia> test(2,1)
ERROR: UndefVarError: relation not defined
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

if 블록도 값을 반환하기 때문에 다른 많은 언어에서 오는 사용자에게는 어색해 보일 수 있습니다. 이 값은 단순히 선택한 분기에서 마지막으로 실행한 명령문의 반환값이며, 따라서

```
julia> x = 3
3

julia> if x > 0
           "positive!"
       else
           "negative..."
       end
"positive!"
```

아주 짧은(한 줄로 된) 조건문은 다음 절에 설명되었듯이 Julia의 단락 회로 계산을 통해 자주 표현된다는 것을 유념하시기 바랍니다.

C, MATLAB, Perl, Python, Ruby와는 다르게 조건식의 값이 true나 false가 아니면 오류가 발생하며, 이는 Java와 같이 자료형을 엄격하게 다루는 언어와 비슷하다고 할 수 있습니다.

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

이 오류는 조건부에 잘못된 자료형을 넣었음을 나타냅니다. `Int64`형 대신 `Bool`형이 들어가야 하죠.

소위 "삼항 연산자"라고 불리는 `?:`는 `if-elseif-else` 구문과 밀접한 관련이 있습니다. 후자가 긴 코드 블록의 조건 실행에 사용되는 것과 달리, 전자는 단일식에서의 조건부 선택이 필요한 곳에서 사용됩니다. 이 연산자는 대부분의 다른 언어에서도 피연산자 셋을 취하는 유일한 연산자라는 칭호를 얻었습니다.

```
a ? b : c
```

`?` 앞의 `a`는 조건식이고, 삼항 연산자는 `a`가 `true`이면 `b`를, `false`이면 `c`를 실행합니다. 여기서 `?`와 `:` 주위에는 공백이 있어야 함을 명심하십시오. `a?b:c`와 같은 식은 유효하지 않은 식입니다.(다만 `?`와 `:` 각각의 뒤에 개행 문자는 사용 가능)

이 동작을 이해하는 가장 쉬운 방법은 예제를 보는 것입니다. 이전 예제에서 `println` 호출은 세 브랜치 모두에서 공유되었습니다. 실제로 고른 것은 오직 출력할 리터럴 문자열이었습니다. 이제 삼항 연산자를 사용하여 보다 간결하게 예제를 작성할 수 있습니다. 명확히 하기 위해, 먼저 둘 중 하나를 고르는 버전을 사용해 봅시다.

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

`x < y` 식이 참이면, 전체 삼항 연산자 식은 `"less than"` 문자열을 계산하고, 거짓이면 `"not less than"` 문자열을 출력할 것입니다. 기존의 셋 중 하나를 고르는 예제를 구현하려면 삼항 연산자를 여러 번 사용하여 중첩할 필요가 있습니다.

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y
```

```
julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

연결을 쉽게 하기 위해 연산자는 오른쪽에서 왼쪽으로 연결됩니다.

`if-elseif-else`와 같이 조건식이 각각 `true`나 `false`로 계산될 때만 `:` 앞뒤로 있는 식이 계산된다는 점 역시 중요합니다.

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"

julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

9.3 단락 계산

단락 계산은 조건부 계산과 상당히 유사합니다. 이 동작은 `&&` 및 `||` 연산자가 있는 대부분의 명령형 프로그래밍 언어에서 찾을 수 있습니다. 이런 연산자로 연결된 일련의 표현식에서, 최종 논리값을 결정하는 데 필요한 최소 식만 계산됩니다. 명쾌하게 말하자면, 이는 다음을 의미합니다.

- 표현식 `a && b`에서, 하위 표현식 `b`는 오직 `a`가 `true`로 계산될 때만 계산을 받는다.
- 표현식 `a || b`에서, 하위 표현식 `b`는 오직 `a`가 `false`로 계산될 때만 계산을 받는다.

왜냐 하면, `a`가 `false`이면, `b`의 값에 관계없이 `a && b`는 무조건 `false`가 되고, `a`가 `true`이면, `b`의 값에 관계없이 `a && b`는 무조건 `true`가 되기 때문입니다. `&&`와 `||` 모두 오른쪽에 연관되지만, `&&`가 `||`보다 우선 순위가 더 높습니다. 실험해보면 동작을 이해하기 쉽습니다.

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)
```

```
julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true

julia> f(1) || f(2)
1
2
false
```

&& 및 || 연산자의 다양한 조합의 연관성과 우선 순위를 통해 같은 방식으로 쉽게 실험할 수 있습니다.

이 동작은 Julia에서 매우 짧은 `if` 문의 대용으로 자주 사용됩니다. `if` <조건> <문장> `end` 대신에, <조건> 그리고 나서 <문장>이라고 읽을 수 있는 <조건> `&&` <문장>을 쓸 수 있습니다. 비슷하게, `if !` <조건> <문장> `end` 대신에, <조건> 아니면 <문장>이라고 읽을 수 있는 <조건> `||` <문장>을 쓸 수 있습니다.

예제로 팩토리얼 함수를 다음과 같이 재귀적으로 선언할 수 있습니다.

```
julia> function fact(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * fact(n-1)
       end
fact (generic function with 1 method)

julia> fact(5)
120

julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fact(::Int64) at ./none:2
 [3] top-level scope
```

단락 계산이 없는 논리 연산은 산술 연산과 기본 함수에서 소개된 비트 논리 연산자 `&`, `|` 로 할 수 있습니다. 이들은 이항연산자 구문을 지원하지만, 항상 인수를 계산하는 일반적인 함수라고 할 수 있습니다.

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

`if`, `elseif` 또는 삼항 연산자에서 사용되는 조건식과 마찬가지로, `&&`나 `||` 역시 피연산자가 논리값(`true` 또는 `false`)을 가져야 합니다. 조건부 체인의 가장 마지막 항목을 제외하고는 어디에도 비논리값을 사용하면 오류가 발생합니다.

```
julia> 1 && true
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

반면에 조건부 체인 끝에는 어떤 표현식이든 사용할 수 있습니다. 이는 선행 조건에 따라 계산되고 반환될 것이기 때문입니다.

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

9.4 반복 계산: 루프

반복 계산은 `while`과 `for` 구문으로 수행합니다. 다음은 `while` 루프의 예제입니다.

```
julia> i = 1;

julia> while i <= 5
           println(i)
           global i += 1
       end
1
2
3
4
5
```

`while`은 조건식(여기서는 `i <= 5`)을 계산하여, `false`일 때까지 `while` 루프를 실행합니다.

`for`은 평범한 특정 횟수를 반복하게 합니다. 위 예제는 `for` 루프로 보다 간결하게 표현할 수 있습니다.

```
julia> for i = 1:5
           println(i)
       end
1
2
```

```
3
4
5
```

여기에서 `1:5`는 범위 객체이며 숫자 1, 2, 3, 4, 5의 순서를 나타냅니다. `for` 루프는 이 값들을 반복하며, 각 수들을 차례로 변수 `i`에 할당합니다. 앞의 `while` 루프 형식과 `for` 루프 형식의 중요한 차이점 중 하나는 바로 변수가 표시되는 범위입니다. 만약 변수 `i`가 다른 영역에서 선언되지 않았다면, `for` 루프 형식에서는 `for` 루프 내부에서만 볼 수 있고, 루프 외부나 루프 종료 이후로는 볼 수 없습니다. 이를 테스트하려면 새로운 대화형 세션 인스턴스나 다른 변수 이름이 필요할 겁니다.

```
julia> for j = 1:5
           println(j)
       end
1
2
3
4
5

julia> j
ERROR: UndefVarError: j not defined
```

변수 범위에 관한 자세한 설명은 Scope of Variables 문서를 통해 확인하십시오. 일반적으로, `for` 루프는 컨테이너 형태의 객체에서도 반복할 수 있습니다. 이 경우, 코드의 더 명확한 가독성을 위해 `=` 대신 `in`이나 `∈`가 대용(그러나 완전히 동등한)으로 사용됩니다.

```
julia> for i in [1,4,0]
           println(i)
       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]
           println(s)
       end
foo
bar
baz
```

다양한 유형의 반복 가능한 컨테이너가 매뉴얼 뒷부분(예: 다차원 배열 문서 참조)에서 소개되고 논의될 것입니다.

실행 중간에 `while`이나 `for` 을 멈추는 것이 편리할 때가 있습니다. 이는 `break` 키워드로 수행할 수 있습니다.

```
julia> i = 1;

julia> while true
           println(i)
           if i >= 5
               break
           end
           global i += 1
       end
1
2
3
4
5

julia> for j = 1:1000
           println(j)
           if j >= 5
               break
           end
       end
1
2
3
4
5
```

위 예제에서는 `break` 키워드 없이는 `while` 루프는 절대 스스로 종료되지 않을 것이며, `for` 루프는 1000까지 세고 말 것입니다. 이 루프 모두 `break`를 사용하여 빠져나갈 수 있습니다.

다른 상황에서는 반복을 중지하고 즉시 다음 단계로 넘어가고 싶을 때 `continue`로 대처할 수 있습니다.

```
julia> for i = 1:10
           if i % 3 != 0
               continue
           end
```

```
            println(i)
        end
3
6
9
```

이는 조건을 무효화하고 `println` 호출을 `if` 블록 안에 두어 더 똑같은 동작을 명확하게 나타낼 수 있기 때문에 다소 고안된 예제입니다. 실제 코드에서는 `continue` 뒤에 계산할 코드가 더 많을 것이며, 종종 `continue`가 여러 번 사용될 수도 있습니다.

다중 중첩 `for` 루프는 하나의 외부 루프로 결합되어, 반복용 변수의 데카르트 곱을 형성합니다.

```
julia> for i = 1:2, j = 3:4
            println((i, j))
        end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

위 문법에서는 범위 객체가 외부 루프 변수의 값을 사용할 수 있습니다(예: `for i = 1:n, j = 1:i`). 그러나 위와 같은 선언에서 `break`는 모든 루프를 탈출하게 됩니다. `i`와 `j` 변수 둘 다 매번 루프가 실행 될 때 값이 지정되므로 실행 중에 `i` 값을 바꿔도 그 다음 루프에서 이를 확인 할 수 없습니다:

```
julia> for i = 1:2, j = 3:4
            println((i, j))
            i = 0
        end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

만약 위 예제가 이중 `for` 루프로 구현된다면 `i`값이 중간에 초기화되지 않아 `0`을 출력하는 것을 볼 수 있습니다.

9.5 예외 처리

예기치 않은 조건이 발생하면 함수가 호출자에게 적절한 값을 반환하지 못할 수 있습니다. 이런 경우에는 예외적인 조건에서 진단 오류 메시지를 출력하는 동안 프로그램을 종료하는 것이 좋을 수도 있지만, 프로그래머가 예외적인 상황을 처리하는 코드를 제공한 경우 해당 코드가 적절한 조치를 취하도록 하는 것이 최선의 방법입니다.

기본 예외 타입

예기치 않은 조건이 일어나면 `Exception`이 발생합니다. 아래에 나열된 기본 제공 `Exception`은 모두 정상적인 제어 흐름을 방해합니다.

Exception
ArgumentError
BoundsError
CompositeException
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
Meta.ParseError
SystemError
TypeError
UndefRefError
UndefVarError
StringIndexError

예를 들어, 음의 실수 값에 적용된 `sqrt` 함수는 `DomainError`를 throw합니다.

```
julia> sqrt(-1)
```

```
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

다음과 같은 방법으로 사용자 정의 예외를 직접 만들 수 있습니다.

```
julia> struct MyCustomException <: Exception end
```

throw 함수

예외는 throw를 사용하여 명시적으로 만들 수 있습니다. 예를 들어, 인수가 음수이면 인수가 음수가 아닌 숫자로만 정의된 함수를 작성하여 DomainError를 throw할 수 있습니다.

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must be nonnegative"))
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError with -1:
argument must be nonnegative
Stacktrace:
 [1] f(::Int64) at ./none:1
```

괄호가 없는 DomainError는 예외가 아니라 예외 타입임을 기억하십시오. Exception 객체를 얻으려면 호출해야 합니다.

```
julia> typeof(DomainError(nothing)) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

또한 일부 예외 유형은 오류 보고에 사용되는 하나 이상의 인수를 필요로 합니다.

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: x not defined
```

이 메커니즘은 UndefVarError가 쓰여지는 방식에 따라 사용자 정의 예외 유형에 의해 쉽게 구현될 수 있습니다.

```
julia> struct MyUndefVarError <: Exception
           var::Symbol
       end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

!!! 주의 오류 메시지를 작성할 때 첫 번째 단어를 소문자로 만드는 것이 좋습니다. 예를 들어, size(A) == size(B) ¦¦ throw(DimensionMismatch("size of A not equal to size of B"))

```
가
 아래보다 더 좋은 예외 처리입니다

`size(A) == size(B) ¦¦ throw(DimensionMismatch("Size of A not equal to size of B"))`.하지만

 때로는 대문자의 첫 번째 문자는 그대로 두는 것이 좋은데, 예를 들어 함수의 인수가 대문자일 경우입니다. `size(A,1) ==
     size(B,2) ¦¦ throw(DimensionMismatch("A has first dimension..."))`.
```

오류

error 함수는 정상적인 제어 흐름을 방해하는 ErrorException을 생성하는 데 사용됩니다.

음수의 제곱근을 취하면 즉시 실행을 멈추고 싶다고 합시다. 이것을 하기 위해 인수가 음수이면 오류가 발생하는 sqrt 함수의 까다로운 버전을 정의할 수 있습니다.

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt(::Int64) at ./none:1
 [3] top-level scope
```

fussy_sqrt가 호출 함수의 실행을 계속하려 하는 것이 아니라 다른 함수에서 음수 값으로 호출되면, 즉시 반환되어 대화식 세션에 오류 메시지를 표시합니다.

```
julia> function verbose_fussy_sqrt(x)
           println("before fussy_sqrt")
           r = fussy_sqrt(x)
           println("after fussy_sqrt")
           return r
       end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt at ./none:1 [inlined]
 [3] verbose_fussy_sqrt(::Int64) at ./none:3
 [4] top-level scope
```

try/catch문

The `try/catch` statement allows for `Exception`s to be tested for, and for the graceful handling of things that may ordinarily break your application. For example, in the below code the function for square root would normally throw an exception. By placing a `try/catch` block around it we can mitigate that here. You may choose how you wish to handle this exception, whether logging it, return a placeholder value or as in the case below where we just printed out a statement. One thing to think about when deciding how to handle unexpected situations is that using a `try/catch` block is much slower than using conditional branching to handle those situations. Below there are more examples of handling exceptions with a `try/catch` block:

```
julia> try
           sqrt("ten")
       catch e
           println("You should have entered a numeric value")
       end
You should have entered a numeric value
```

또한 try/catch문은 Exception이 변수에 저장되도록 합니다. 이 고안된 예제에서, 다음 예제는 x가 색인 가능한 경우 x의 두 번째 요소의 제곱근을 계산하고, 그렇지 않으면 x가 실수임을 가정하고 제곱근을 반환합니다.

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           end
       end
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])
2.0

julia> sqrt_second([1 -4])
0.0 + 2.0im

julia> sqrt_second(9)
3.0

julia> sqrt_second(-9)
ERROR: DomainError with -9.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

catch 다음의 기호는 항상 예외 이름으로 해석될 것이고, 때문에 한 줄로 try/catch문을 작성할 때 주의해야 합니다. 다음 코드는 오류가 발생하더라도 x의 값을 반환하지 않습니다.

```
try bad() catch x end
```

대신 세미콜론을 사용하거나 catch 다음에 개행 문자를 삽입하십시오.

```
try bad() catch; x end


try bad()
```

```
catch
    x
end
```

try/catch문의 장점은 호출 함수의 스택에서 훨씬 더 높은 레벨로 깊게 중첩된 계산을 즉시 풀 수 있는 능력에 있습니다. 오류가 발생하지 않은 상황이 있지만 스택을 풀어 더 높은 레벨로 값을 전달하는 것이 바람직합니다. Julia는 고급 오류 처리를 위해 rethrow, backtrace, catch_backtrace 그리고 Base.catch_stack와 같은 함수들을 제공합니다.

finally문

상태 변경을 수행하거나 파일과 같은 리소스를 사용하는 코드에서는 일반적으로 코드가 끝났을 때 수행해야 하는 정리 작업(예: 파일 닫기)이 있습니다. 예외는 이 작업을 어쩌면 복잡하게 만들 수 있습니다. 왜냐하면 코드 블록이 정상적으로 끝나기 전에 종료될 수 있기 때문입니다. finally 키워드는 주어진 코드 블록의 종료가 정상 유무를 가리지 않고 특정 코드를 실행하게 해줍니다.

예를 들면, 열린 파일을 확실히 닫을 수 있는 방법은 다음과 같습니다.

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

제어가 try 블록을 떠날 때(return 때문에 끝나든, 정상적으로 끝나든) close(f)가 실행됩니다. 만약 여기서 try 블록이 예외로 인해 종료되면 예외는 계속 증식할 것입니다. catch 블록은 try 및 finally와 결합할 수 있으므로, 이 상황에서는 catch가 오류를 처리한 후에 finally문이 실행되면 좋을 것입니다.

9.6 태스크 (일명 코루틴)

태스크는 유연한 방식으로 계산을 일시 중단하고 다시 시작할 수 있게 해주는 제어 흐름 기능입니다. 이 기능은 다른 프로그래밍 언어에서는 대칭 코루틴, 경량 스레드, 협업 멀티태스킹 또는 원샷 컨티뉴에이션과 같은 다른 이름으로 불립니다.

컴퓨팅 작업(실제로는 특정 기능 실행)이 Task로 지정되면, 다른 Task로 전환하여 그 태스크를 중단할 수 있습니다. 원래의 Task는 나중에 다시 시작될 수 있으며, 중단된 그 시점에서 바로 시작됩니다. 처음에는 함수 호출과 비슷하게 보일 수 있지만, 두 가지 중요한 차이점이 있습니다. 첫째, 태스크 전환은 공간을 사용하지 않아 호출 스택을 사용하지 않고도 얼마든지 태스크 전환이 발생할 수 있습니다. 둘째, 함수 호출과는 달리 태스크간 전환은 임의의 순서로 발생할 수 있습니다. 함수 호출은 호출된 함수가 제어가 호출 함수로 돌아가기 전에 실행을 완료해야 하는 구조입니다.

이러한 종류의 제어 흐름은 특정 문제를 훨씬 쉽게 해결할 수 있습니다. 일부 문제에서 필요한 작업의 다양한 부분은 함수 호출에 의해 자연스럽게 관련되지 않습니다. 수행해야 할 작업 중에 명확한 "호출자"나 "호출 수신자"가 없기 때문입니다. 한 예로 복잡한 프로시저가 값을 생성하고, 다른 복잡한 프로시저가 값을 소비하는 생산자-소비자 문제가 있습니다. 소비자는 단순히 값을 얻기 위해 생산자 함수를 호출할 수 없습니다. 왜냐하면 생산자가 생성할 값이 더 많아 반환할 준비가 되지 않았기 때문입니다. 태스크를 통해 생산자와 소비자는 필요한 만큼 오래 실행하고 필요한 만큼 값을 주고 받을 수 있습니다.

Julia는 이 문제를 해결하기 위한 `Channel` 메커니즘을 제공합니다. `Channel`은 여러 태스크를 읽고 쓸 수 있는 대기 가능한 선입선출(FIFO) 대기열(queue)입니다.

`put!` 호출을 통해 값을 생성하는 생산자 태스크를 정의해 봅시다. 값을 소비하려면 생산자가 새 태스크를 실행하도록 예약해야 합니다. 인수가 하나인 함수를 인수로 받아들이는 특별한 `Channel` 생성자는 채널에 묶여진 작업을 실행하는 데 사용할 수 있습니다. 그런 다음 채널 객체에서 반복적으로 값을 `take!`를 통해 가져올 수 있습니다.

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:4
               put!(c, 2n)
           end
           put!(c, "stop")
       end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"

julia> take!(chnl)
2

julia> take!(chnl)
4

julia> take!(chnl)
6

julia> take!(chnl)
8

julia> take!(chnl)
"stop"
```

이 동작을 생각하는 한 가지 방법은 producer가 여러 번 반환이 가능하다는 것입니다. put! 호출 사이에 생성자의 실행이 일시 중단되고 소비자가 제어권을 가집니다.

반환된 Channel은 for 루프에서 반복용 객체로 사용될 수 있습니다. 이때 루프 변수는 생성된 모든 값을 취합니다. 채널이 닫히면 루프도 종료됩니다.

```
julia> for x in Channel(producer)
           println(x)
       end
start
2
4
6
8
stop
```

생산자 측에서 채널을 명시적으로 닫을 필요는 없었음을 알아두십시오. 이는 채널을 태스크에 묶는 동작이 채널의 수명과 묶인 태스크의 수명을 연결짓기 때문입니다. 채널 객체는 태스크가 종료되면 자동으로 닫힙니다. 여러 채널을 하나의 태스크에 묶을 수 있고, 그 반대로도 가능합니다.

태스크 생성자가 인수가 없는 함수를 예상하는 동안, 태스크에 묶인 채널을 만드는 채널 메소드는 채널 유형의 단일 인수를 허용하는 함수를 필요로 합니다. 공통 패턴은 생산자가 매개 변수화된 경우이며, 이 경우 부분 함수 응용 프로그램은 인수가 없거나 1개의 인수를 갖는 익명 함수를 작성하는 데 필요합니다.

태스크 객체의 경우 직접 또는 편리한 매크로를 사용하여 수행할 수 있습니다.

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# 또는 동일하게
taskHdl = @task mytask(7)
```

고급 작업 배분 패턴을 조율하기 위해, 태스크 및 채널 생성자와 바인드 및 스케쥴을 사용하여 일련의 채널을 생산자/소비자 태스크 집합과 명시적으로 연결할 수 있습니다.

현재 Julia의 태스크 기능은 별도의 CPU 코어를 사용하도록 스케쥴되어 있지 않습니다. 진짜 커널 스레드는 Parallel Computing 주제에서 논의하겠습니다.

코어 태스크 연산

작업 중 태스크가 어떻게 전환되는지 이해하기 위해 `yieldto` 저수준 구조를 탐험해 봅시다. `yieldto(task,value)`는 현재 태스크를 잠시 중단하고, 지정된 태스크로 전환하며, 태스크가 특정한 값을 반환하도록 하는 `yieldto` 호출을 하도록 합니다. 태스크 방식 제어 흐름을 사용하려면 `yieldto`만이 유일한 해법임을 명심하십시오. 호출하고 반환하는 동작 대신 항상 다른 태스크로 전환할 뿐입니다. 이것이 이 기능이 "대칭 코루틴"이라고 불리는 이유입니다. 각 태스크는 동일한 메커니즘을 통해 다른 태스크로 전환하거나 전환됩니다.

`yieldto`는 강력하지만, 하지만 대부분의 태스크가 그것을 직접 호출하지는 않습니다. 왜 그런지 한번 볼까요. 현재 태스크를 전환한다면 아마 특정 시점에서 다시 전환하고 싶을 겁니다. 하지만 언제 전환을 해야 하는지, 무슨 태스크를 전환해야할 지를 파악하는 데에 상당한 조율이 필요할 것입니다. 예를 들어, `put!`과 `take!`는 채널 컨텍스트에서 사용될 때 소비자가 누구인지를 기억하기 위해 상태를 유지하는 차단 작업입니다. 소비 작업을 수동으로 추적할 필요가 없으므로 `put!`을 저수준인 `yieldto`보다 쉽게 사용할 수 있습니다.

`yieldto` 외에, 태스크를 효과적으로 사용하기 위해서는 몇 가지 기본적인 함수가 더 필요합니다.

- `current_task`는 현재 실행중인 태스크를 참조합니다.
- `istaskdone`는 태스크가 종료했는지 여부를 묻습니다.
- `istaskstarted`는 태스크가 실행 중인지 여부를 묻습니다.
- `task_local_storage` 현재 태스크와 관련된 키값 저장소를 조작합니다.

태스크와 이벤트

대부분의 태스크 전환은 입출력 요청과 같은 이벤트를 기다린 결과로 발생하며, 이는 줄리아 Base에 포함된 스케줄러에 의해 수행됩니다. 스케줄러는 실행 가능한 작업 대기열을 관리하고 메시지 도착과 같은 외부 이벤트를 기반으로 작업을 다시 시작하는 이벤트 루프를 실행합니다.

이벤트를 기다리는 기본적인 함수로는 `wait`가 있습니다. 여러 객체는 `wait`를 구현할 수 있는데, 그 예로 `Process` 객체가 주어진다면, `wait`는 그 객체가 종료될 때까지 기다릴 것입니다. `wait`는 종종 명시적이지 않습니다. 예를 들자면, `wait`는 데이터를 사용할 수 있을 때까지 기다리기 위해 `read` 호출 내부에서 발생할 수 있습니다.

이 모든 경우에, `wait`는 태스크를 대기열에 넣고 재시작하는 `Condition` 객체에서 궁극적으로 작동합니다. 태스크가 `Condition`에서 `wait`를 호출하면, 태스크는 실행 불가능한 것으로 표시되고, 조건 대기열에 추가되며 스케줄러로 전환됩니다. 스케줄러는 실행할 다른 태스크를 선택하거나 외부 이벤트 대기를 차단합니다. 모든 것이 잘되면 결국 이벤트 처리기가 조건에서 `notify`를 호출하여 해당 조건을 기다리는 작업을 다시 실행 가능하게 만듭니다.

태스크를 호출하여 명시적으로 생성된 태스크는 처음에는 스케줄러에게 알려지지 않습니다. 원한다면 `yieldto`를 사용하여 수동으로 작업을 관리할 수도 있습니다. 하지만 그런 태스크가 이벤트를 기다리면 예상대로 이벤트가 발생할 때 자동적으로 재시작됩니다. 이벤트를 기다리지 않고 언제든지 스케줄러가 작업을 실행할 수 있게 하는 것도 가능합니다. 이 방법은 `schedule`을 호출하거나, `@async` 매크로를 사용하여 수행할 수 있습니다 (자세한 사항은 Parallel Computing 참조).

태스크 상태

태스크에는 실행 상태를 설명하는 `state` 필드가 있습니다. 태스크의 모든 `state`는 다음과 같습니다.

상태	의미
`:runnable`	현재 실행 중이거나 실행 가능한 상태
`:done`	실행을 성공적으로 완료한 상태
`:failed`	알 수 없는 예외로 종료된 상태

Chapter 10

Scope of Variables

The scope of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called `x` without the two `x`'s referring to the same thing. Similarly, there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce scope blocks, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in Julia, global scope and local scope. The latter can be nested. The constructs introducing scope blocks are:

Scope constructs

Construct	Scope type	Scope blocks it may be nested in
`module`, `baremodule`	global	global
interactive prompt (REPL)	global	global
(mutable) `struct`, `macro`	local	global
`for`, `while`, `try-catch-finally`, `let`	local	global or local
functions (either syntax, anonymous & do-blocks)	local	global or local
comprehensions, broadcast-fusing	local	global or local

Notably missing from this table are begin blocks and if blocks which do not introduce new scopes. Both types of scopes follow somewhat different rules which will be explained below.

Julia uses lexical scoping, meaning that a function's scope does not inherit from its caller's scope, but from the scope

in which the function was defined. For example, in the following code the `x` inside `foo` refers to the `x` in the global scope of its module `Bar`:

```
julia> module Bar
           x = 1
           foo() = x
       end;
```

and not a `x` in the scope where `foo` is used:

```
julia> import .Bar

julia> x = -1;

julia> Bar.foo()
1
```

Thus lexical scope means that the scope of variables can be inferred from the source code alone.

10.1 Global Scope

Each module introduces a new global scope, separate from the global scope of all other modules; there is no all-encompassing global scope. Modules can introduce variables of other modules into their scope through the using or import statements or through qualified access using the dot-notation, i.e. each module is a so-called namespace. Note that variable bindings can only be changed within their global scope and not from an outside module.

```
julia> module A
           a = 1 # a global in A's scope
       end;

julia> module B
           module C
               c = 2
           end
           b = C.c    # can access the namespace of a nested global scope
                      # through a qualified access
           import ..A # makes module A available
           d = A.a
       end;
```

```
julia> module D
           b = a # errors as D's global scope is separate from A's
       end;
ERROR: UndefVarError: a not defined

julia> module E
           import ..A # make module A available
           A.a = 2    # throws below error
       end;
ERROR: cannot assign variables in other modules
```

Note that the interactive prompt (aka REPL) is in the global scope of the module `Main`.

10.2 Local Scope

A new local scope is introduced by most code blocks (see above table for a complete list). A local scope inherits all the variables from a parent local scope, both for reading and writing. Unlike global scopes, local scopes are not namespaces, thus variables in an inner scope cannot be retrieved from the parent scope through some sort of qualified access.

The following rules and examples pertain to local scopes. A newly introduced variable in a local scope cannot be referenced by a parent scope. For example, here the z is not introduced into the top-level scope:

```
julia> for i = 1:10
           z = i
       end

julia> z
ERROR: UndefVarError: z not defined
```

> Note
>
> In this and all following examples it is assumed that their top-level is a global scope with a clean workspace, for instance a newly started REPL.

Inner local scopes can, however, update variables in their parent scopes:

```
julia> for i = 1:1
           z = i
```

```
            for j = 1:1
                z = 0
            end
            println(z)
        end
0
```

Inside a local scope a variable can be forced to be a new local variable using the `local` keyword:

```
julia> for i = 1:1
            x = i + 1
            for j = 1:1
                local x = 0
            end
            println(x)
        end
2
```

Inside a local scope a global variable can be assigned to by using the keyword `global`:

```
julia> for i = 1:10
            global z
            z = i
        end

julia> z
10
```

The location of both the **`local`** and **`global`** keywords within the scope block is irrelevant. The following is equivalent to the last example (although stylistically worse):

```
julia> for i = 1:10
            z = i
            global z
        end

julia> z
10
```

The `local` and `global` keywords can also be applied to destructuring assignments, e.g. `local x, y = 1, 2`. In this case the keyword affects all listed variables.

In a local scope, all variables are inherited from its parent global scope block unless:

- an assignment would result in a modified global variable, or
- a variable is specifically marked with the keyword `local`.

Thus global variables are only inherited for reading, not for writing:

```
julia> x, y = 1, 2;

julia> function foo()
           x = 2        # assignment introduces a new local
           return x + y # y refers to the global
       end;

julia> foo()
4

julia> x
1
```

An explicit `global` is needed to assign to a global variable:

> Avoiding globals
>
> Avoiding changing the value of global variables is considered by many to be a programming best-practice. Changing the value of a global variable can cause "action at a distance", making the behavior of a program harder to reason about. This is why the scope blocks that introduce local scope require the `global` keyword to declare the intent to modify a global variable.

```
julia> x = 1;

julia> function foobar()
           global x = 2
       end;

julia> foobar();
```

```
julia> x
2
```

Note that nested functions can modify their parent scope's local variables:

```
julia> x, y = 1, 2;

julia> function baz()
           x = 2 # introduces a new local
           function bar()
               x = 10       # modifies the parent's x
               return x + y # y is global
           end
           return bar() + x # 12 + 10 (x is modified in call of bar())
       end;

julia> baz()
22

julia> x, y # verify that global x and y are unchanged
(1, 2)
```

The reason to allow modifying local variables of parent scopes in nested functions is to allow constructing `closures` which have private state, for instance the **`state`** variable in the following example:

```
julia> let state = 0
           global counter() = (state += 1)
       end;

julia> counter()
1

julia> counter()
2
```

See also the closures in the examples in the next two sections. A variable, such as `x` in the first example and **`state`** in the second, that is inherited from the enclosing scope by the inner function is sometimes called a captured variable. Captured variables can present performance challenges discussed in performance tips.

The distinction between inheriting global scope and nesting local scope can lead to some slight differences between functions defined in local versus global scopes for variable assignments. Consider the modification of the last example by moving `bar` to the global scope:

```
julia> x, y = 1, 2;

julia> function bar()
           x = 10 # local, no longer a closure variable
           return x + y
       end;

julia> function quz()
           x = 2 # local
           return bar() + x # 12 + 2 (x is not modified)
       end;

julia> quz()
14

julia> x, y # verify that global x and y are unchanged
(1, 2)
```

Note that the above nesting rules do not pertain to type and macro definitions as they can only appear at the global scope. There are special scoping rules concerning the evaluation of default and keyword function arguments which are described in the Function section.

An assignment introducing a variable used inside a function, type or macro definition need not come before its inner usage:

```
julia> f = y -> y + a;

julia> f(3)
ERROR: UndefVarError: a not defined
Stacktrace:
[...]

julia> a = 1
1
```

```
julia> f(3)
4
```

This behavior may seem slightly odd for a normal variable, but allows for named functions – which are just normal variables holding function objects – to be used before they are defined. This allows functions to be defined in whatever order is intuitive and convenient, rather than forcing bottom up ordering or requiring forward declarations, as long as they are defined by the time they are actually called. As an example, here is an inefficient, mutually recursive way to test if positive integers are even or odd:

```
julia> even(n) = (n == 0) ? true : odd(n - 1);

julia> odd(n) = (n == 0) ? false : even(n - 1);

julia> even(3)
false

julia> odd(3)
true
```

Julia provides built-in, efficient functions to test for oddness and evenness called `iseven` and `isodd` so the above definitions should only be considered to be examples of scope, not efficient design.

Let Blocks

Unlike assignments to local variables, **`let`** statements allocate new variable bindings each time they run. An assignment modifies an existing value location, and **`let`** creates new locations. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The **`let`** syntax accepts a comma-separated series of assignments and variable names:

```
julia> x, y, z = -1, -1, -1;

julia> let x = 1, z
           println("x: $x, y: $y") # x is local variable, y the global
           println("z: $z") # errors as z has not been assigned yet but is local
       end
x: 1, y: -1
ERROR: UndefVarError: z not defined
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x` since the two `x` variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           Fs[i] = ()->i
           global i += 1
       end

julia> Fs[1]()
3

julia> Fs[2]()
3
```

Here we create and store two closures that return variable `i`. However, it is always the same variable `i`, so the two closures behave identically. We can use `let` to create a new binding for `i`:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           let i = i
               Fs[i] = ()->i
           end
           global i += 1
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Since the `begin` construct does not introduce a new scope, it can be useful to use a zero-argument `let` to just introduce a new scope block without creating any new bindings:

```
julia> let
           local x = 1
```

```
            let
                local x = 2
            end
            x
        end
1
```

Since `let` introduces a new scope block, the inner local `x` is a different variable than the outer local `x`.

For Loops and Comprehensions

`for` loops, `while` loops, and Comprehensions have the following behavior: any new variables introduced in their body scopes are freshly allocated for each loop iteration, as if the loop body were surrounded by a `let` block:

```
julia> Fs = Vector{Any}(undef, 2);

julia> for j = 1:2
            Fs[j] = ()->j
        end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

A `for` loop or comprehension iteration variable is always a new variable:

```
julia> function f()
            i = 0
            for i = 1:3
            end
            return i
        end;

julia> f()
0
```

However, it is occasionally useful to reuse an existing local variable as the iteration variable. This can be done conveniently by adding the keyword `outer`:

```
julia> function f()
           i = 0
           for outer i = 1:3
           end
           return i
       end;

julia> f()
3
```

10.3 Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the `const` keyword:

```
julia> const e  = 2.71828182845904523536;

julia> const pi = 3.14159265358979323846;
```

Multiple variables can be declared in a single `const` statement:

```
julia> const a, b = 1, 2
(1, 2)
```

The `const` declaration should only be used in global scope on globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a `const` declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary, and in fact are currently not supported.

Special top-level assignments, such as those performed by the `function` and `struct` keywords, are constant by default.

Note that `const` only affects the variable binding; the variable may be bound to a mutable object (such as an array), and that object may still be modified. Additionally when one tries to assign a value to a variable that is declared constant the following scenarios are possible:

- if a new value has a different type than the type of the constant then an error is thrown:

```
julia> const x = 1.0
1.0

julia> x = 1
ERROR: invalid redefinition of constant x
```

- if a new value has the same type as the constant then a warning is printed:

```
julia> const y = 1.0
1.0

julia> y = 2.0
WARNING: redefining constant y
2.0
```

- if an assignment would not result in the change of variable value no message is given:

```
julia> const z = 100
100

julia> z = 100
100
```

The last rule applies for immutable objects even if the variable binding would change, e.g.:

```
julia> const s1 = "1"
"1"

julia> s2 = "1"
"1"

julia> pointer.([s1, s2], 1)
2-element Array{Ptr{UInt8},1}:
 Ptr{UInt8} @0x00000000132c9638
 Ptr{UInt8} @0x0000000013dd3d18

julia> s1 = s2
"1"
```

```
julia> pointer.([s1, s2], 1)
2-element Array{Ptr{UInt8},1}:
 Ptr{UInt8} @0x0000000013dd3d18
 Ptr{UInt8} @0x0000000013dd3d18
```

However, for mutable objects the warning is printed as expected:

```
julia> const a = [1]
1-element Array{Int64,1}:
 1

julia> a = [1]
WARNING: redefining constant a
1-element Array{Int64,1}:
 1
```

Note that although sometimes possible, changing the value of a `const` variable is strongly discouraged, and is intended only for convenience during interactive use. Changing constants can cause various problems or unexpected behaviors. For instance, if a method references a constant and is already compiled before the constant is changed then it might keep using the old value:

```
julia> const x = 1
1

julia> f() = x
f (generic function with 1 method)

julia> f()
1

julia> x = 2
WARNING: redefining constant x
2

julia> f()
1
```

Chapter 11

Types

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in 메서드, but is rooted in the type system presented here.

The default behavior in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia functions without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Adding annotations serves three primary purposes: to take advantage of Julia's powerful multiple-dispatch mechanism, to improve human readability, and to catch programmer errors.

Describing Julia in the lingo of type systems, it is: dynamic, nominative and parametric. Generic types can be parameterized, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. Other high-level aspects of Julia's type system that

should be mentioned up front are:

- There is no division between object and non-object values: all values in Julia are true objects having a type that belongs to a single, fully connected type graph, all nodes of which are equally first-class as types.
- There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.
- Only values, not variables, have types – variables are simply names bound to values.
- Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols, by values of any type for which `isbits` returns true (essentially, things like numbers and bools that are stored like C types or `struct`s with no pointers to other objects), and also by tuples thereof. Type parameters may be omitted when they do not need to be referenced or restricted.

Julia's type system is designed to be powerful and expressive, yet clear, intuitive and unobtrusive. Many Julia programmers may never feel the need to write code that explicitly uses types. Some kinds of programming, however, become clearer, simpler, faster and more robust with declared types.

11.1 Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect,
2. To provide extra type information to the compiler, which can then improve performance in some cases

When appended to an expression computing a value, the `::` operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation – recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got Int64
```

```
julia> (1+2)::Int
3
```

This allows a type assertion to be attached to any expression in-place.

When appended to a variable on the left-hand side of an assignment, or as part of a `local` declaration, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using `convert`:

```
julia> function foo()
           x::Int8 = 100
           x
       end
foo (generic function with 1 method)

julia> foo()
100

julia> typeof(ans)
Int8
```

This feature is useful for avoiding performance "gotchas" that could occur if one of the assignments to a variable changed its type unexpectedly.

This "declaration" behavior only occurs in specific contexts:

```
local x::Int8  # in a local declaration
x::Int8 = 10   # as the left-hand side of an assignment
```

and applies to the whole current scope, even before the declaration. Currently, type declarations cannot be used in global scope, e.g. in the REPL, since Julia does not yet have constant-type globals.

Declarations can also be attached to function definitions:

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to `Float64`.

11.2 Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

Recall that in Integers and Floating-Point Numbers, we introduced a variety of concrete types of numeric values: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `Float16`, `Float32`, and `Float64`. Although they have different representation sizes, `Int8`, `Int16`, `Int32`, `Int64` and `Int128` all have in common that they are signed integer types. Likewise `UInt8`, `UInt16`, `UInt32`, `UInt64` and `UInt128` are all unsigned integer types, while `Float16`, `Float32` and `Float64` are distinct in being floating-point types rather than integers. It is common for a piece of code to make sense, for example, only if its arguments are some kind of integer, but not really depend on what particular kind of integer. For example, the greatest common denominator algorithm works for all kinds of integers, but will not work for floating-point numbers. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the `abstract type` keyword. The general syntaxes for declaring an abstract type are:

```
abstract type «name» end
abstract type «name» <: «supertype» end
```

The `abstract type` keyword introduces a new abstract type, whose name is given by `«name»`. This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

When no supertype is given, the default supertype is `Any` – a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, `Any` is commonly called "top" because it is at the apex of the type graph. Julia also has a predefined abstract "bottom" type, at the nadir of the type graph, which is written as `Union{}`. It is the exact opposite of `Any`: no object is an instance of `Union{}` and all types are supertypes of `Union{}`.

Let's consider some of the abstract types that make up Julia's numerical hierarchy:

```
abstract type Number end
abstract type Real     <: Number end
```

```
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end
```

The `Number` type is a direct child type of `Any`, and `Real` is its child. In turn, `Real` has two children (it has more, but only two are shown here; we'll get to the others later): `Integer` and `AbstractFloat`, separating the world into representations of integers and representations of real numbers. Representations of real numbers include, of course, floating-point types, but also include other types, such as rationals. Hence, `AbstractFloat` is a proper subtype of `Real`, including only floating-point representations of real numbers. Integers are further subdivided into `Signed` and `Unsigned` varieties.

The `<:` operator in general means "is a subtype of", and, used in declarations like this, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
function myplus(x,y)
    x+y
end
```

The first thing to note is that the above argument declarations are equivalent to `x::Any` and `y::Any`. When this function is invoked, say as `myplus(2,5)`, the dispatcher chooses the most specific method named `myplus` that matches the given arguments. (See 메서드 for more information on multiple dispatch.)

Assuming no method more specific than the above is found, Julia next internally defines and compiles a method called `myplus` specifically for two `Int` arguments based on the generic function given above, i.e., it implicitly defines and compiles:

```
function myplus(x::Int,y::Int)
    x+y
end
```

and finally, it invokes this specific method.

Thus, abstract types allow programmers to write generic functions that can later be used as the default method by many combinations of concrete types. Thanks to multiple dispatch, the programmer has full control over whether the default or more specific method is used.

An important point to note is that there is no loss in performance if the programmer relies on a function whose arguments are abstract types, because it is recompiled for each tuple of argument concrete types with which it is invoked. (There may be a performance issue, however, in the case of function arguments that are containers of abstract types; see Performance Tips.)

11.3 Primitive Types

A primitive type is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Unlike most languages, Julia lets you declare your own primitive types, rather than providing only a fixed set of built-in ones. In fact, the standard primitive types are all defined in the language itself:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

The general syntaxes for declaring a primitive type are:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

The number of bits indicates how much storage the type requires and the name gives the new type a name. A primitive type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having `Any` as its immediate supertype. The declaration of `Bool` above therefore means that a boolean value takes eight bits to store, and has `Integer` as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

The types `Bool`, `Int8` and `UInt8` all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. A fundamental difference between them is that they have different supertypes: `Bool`'s direct supertype is `Integer`, `Int8`'s is `Signed`, and `UInt8`'s is `Unsigned`. All other differences between `Bool`, `Int8`, and `UInt8` are matters of behavior – the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make `Bool` behave any differently than `Int8` or `UInt8`.

11.4 Composite Types

Composite types are called records, structs, or objects in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind of user-definable type, and they are by far the most commonly used user-defined type in Julia as well.

In mainstream object oriented languages, such as C++, Java, Python and Ruby, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as Ruby or Smalltalk, all values are objects whether they are composites or not. In less pure object oriented languages, including C++ and Java, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In Julia, all values are objects, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a function to use by multiple dispatch, meaning that the types of all of a function's arguments are considered when selecting a method, rather than just the first one (see 메서드 for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods into function objects rather than having named bags of methods "inside" each object ends up being a highly beneficial aspect of the language design.

Composite types are introduced with the `struct` keyword followed by a block of field names, optionally annotated with types using the `::` operator:

```
julia> struct Foo
           bar
           baz::Int
```

```
        qux::Float64
    end
```

Fields with no type annotation default to `Any`, and can accordingly hold any type of value.

New objects of type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

When a type is applied like a function it is called a constructor. Two constructors are generated automatically (these are called default constructors). One accepts any arguments and calls `convert` to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Since the `bar` field is unconstrained in type, any value will do. However, the value for `baz` must be convertible to `Int`:

```
julia> Foo((), 23.5, 1)
ERROR: InexactError: Int64(23.5)
Stacktrace:
[...]
```

You may find a list of field names using the `fieldnames` function.

```
julia> fieldnames(Foo)
(:bar, :baz, :qux)
```

You can access the field values of a composite object using the traditional `foo.bar` notation:

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

Composite objects declared with **struct** are immutable; they cannot be modified after construction. This may seem odd at first, but it has several advantages:

- It can be more efficient. Some structs can be packed efficiently into arrays, and in some cases the compiler is able to avoid allocating immutable objects entirely.
- It is not possible to violate the invariants provided by the type's constructors.
- Code using immutable objects can be easier to reason about.

An immutable object might contain mutable objects, such as arrays, as fields. Those contained objects will remain mutable; only the fields of the immutable object itself cannot be changed to point to different objects.

Where required, mutable composite objects can be declared with the keyword `mutable struct`, to be discussed in the next section.

Immutable composite types with no fields are singletons; there can be only one instance of such types:

```
julia> struct NoFields
       end

julia> NoFields() === NoFields()
true
```

The `===` function confirms that the "two" constructed instances of **NoFields** are actually one and the same. Singleton types are described in further detail below.

There is much more to say about how instances of composite types are created, but that discussion depends on both Parametric Types and on 메서드, and is sufficiently important to be addressed in its own section: Constructors.

11.5 Mutable Composite Types

If a composite type is declared with **mutable struct** instead of **struct**, then instances of it can be modified:

```
julia> mutable struct Bar
           baz
           qux::Float64
       end

julia> bar = Bar("Hello", 1.5);
```

```
julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

In order to support mutation, such objects are generally allocated on the heap, and have stable memory addresses. A mutable object is like a little container that might hold different values over time, and so can only be reliably identified with its address. In contrast, an instance of an immutable type is associated with specific field values -- the field values alone tell you everything about the object. In deciding whether to make a type mutable, ask whether two instances with the same field values would be considered identical, or if they might need to change independently over time. If they would be considered identical, the type should probably be immutable.

To recap, two essential properties define immutability in Julia:

- It is not permitted to modify the value of an immutable type.
  - For bits types this means that the bit pattern of a value once set will never change and that value is the identity of a bits type.
  - For composite types, this means that the identity of the values of its fields will never change. When the fields are bits types, that means their bits will never change, for fields whose values are mutable types like arrays, that means the fields will always refer to the same mutable value even though that mutable value's content may itself be modified.
- An object with an immutable type may be copied freely by the compiler since its immutability makes it impossible to programmatically distinguish between the original object and a copy.
  - In particular, this means that small enough immutable values like integers and floats are typically passed to functions in registers (or stack allocated).
  - Mutable values, on the other hand are heap-allocated and passed to functions as pointers to heap-allocated values except in cases where the compiler is sure that there's no way to tell that this is not what is happening.

11.6 Declared Types

The three kinds of types (abstract, primitive, composite) discussed in the previous sections are actually all closely related. They share the same key properties:

- They are explicitly declared.
- They have names.
- They have explicitly declared supertypes.
- They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept, `DataType`, which is the type of any of these types:

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

A `DataType` may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a primitive type is a `DataType` with nonzero size, but no field names. A composite type is a `DataType` that has field names or is empty (zero size).

Every concrete value in the system is an instance of some `DataType`.

11.7 Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special `Union` keyword:

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got Float64
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer. The Julia compiler is able to generate efficient code in the presence of `Union` types with a small number of types [1], by generating specialized code in separate branches for each possible type.

A particularly useful case of a `Union` type is `Union{T, Nothing}`, where `T` can be any type and `Nothing` is the singleton type whose only instance is the object `nothing`. This pattern is the Julia equivalent of `Nullable`, `Option` or `Maybe` types in other languages. Declaring a function argument or a field as `Union{T, Nothing}` allows setting it either to a value of type `T`, or to `nothing` to indicate that there is no value. See this FAQ entry for more information.

11.8 Parametric Types

An important and powerful feature of Julia's type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types – one for each possible combination of parameter values. There are many languages that support some version of generic programming, wherein data structures and algorithms to manipulate them may be specified without specifying the exact types involved. For example, some form of generic programming exists in ML, Haskell, Ada, Eiffel, C++, Java, C#, F#, and Scala, just to name a few. Some of these languages support true parametric polymorphism (e.g. ML, Haskell, Scala), while others support ad-hoc, template-based styles of generic programming (e.g. C++, Java). With so many different varieties of generic programming and parametric types in various languages, we won't even attempt to compare Julia's parametric types to other languages, but will instead focus on explaining Julia's system in its own right. We will note, however, that because Julia is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

All declared types (the `DataType` variety) can be parameterized, with the same syntax in each case. We will discuss them in the following order: first, parametric composite types, then parametric abstract types, and finally parametric primitive types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
julia> struct Point{T}
           x::T
           y::T
       end
```

This declaration defines a new parametric type, `Point{T}`, holding two "coordinates" of type `T`. What, one may ask, is `T`? Well, that's precisely the point of parametric types: it can be any type at all (or a value of any bits type, actually, although here it's clearly used as a type). `Point{Float64}` is a concrete type equivalent to the type defined

by replacing `T` in the definition of `Point` with `Float64`. Thus, this single declaration actually declares an unlimited number of types: `Point{Float64}`, `Point{AbstractString}`, `Point{Int64}`, etc. Each of these is now a usable concrete type:

```
julia> Point{Float64}
Point{Float64}

julia> Point{AbstractString}
Point{AbstractString}
```

The type `Point{Float64}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{AbstractString}` is a "point" whose "coordinates" are string objects (see Strings).

`Point` itself is also a valid type object, containing all instances `Point{Float64}`, `Point{AbstractString}`, etc. as subtypes:

```
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

Other types, of course, are not subtypes of it:

```
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Concrete `Point` types with different values of `T` are never subtypes of each other:

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

> Warning
>
> This last point is very important: even though `Float64 <: Real` we DO NOT have `Point{Float64} <: Point{Real}`.

In other words, in the parlance of type theory, Julia's type parameters are invariant, rather than being covariant (or even contravariant). This is for practical reasons: while any instance of `Point{Float64}` may conceptually be like an instance of `Point{Real}` as well, the two types have different representations in memory:

- An instance of `Point{Float64}` can be represented compactly and efficiently as an immediate pair of 64-bit values;
- An instance of `Point{Real}` must be able to hold any pair of instances of `Real`. Since objects that are instances of `Real` can be of arbitrary size and structure, in practice an instance of `Point{Real}` must be represented as a pair of pointers to individually allocated `Real` objects.

The efficiency gained by being able to store `Point{Float64}` objects with immediate values is magnified enormously in the case of arrays: an `Array{Float64}` can be stored as a contiguous memory block of 64-bit floating-point values, whereas an `Array{Real}` must be an array of pointers to individually allocated `Real` objects – which may well be boxed 64-bit floating-point values, but also might be arbitrarily large, complex objects, which are declared to be implementations of the `Real` abstract type.

Since `Point{Float64}` is not a subtype of `Point{Real}`, the following method can't be applied to arguments of type `Point{Float64}`:

```
function norm(p::Point{Real})
    sqrt(p.x^2 + p.y^2)
end
```

A correct way to define a method that accepts all arguments of type `Point{T}` where `T` is a subtype of `Real` is:

```
function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

(Equivalently, one could define `function norm(p::Point{T} where T<:Real)` or `function norm(p::Point{T}) where T<:Real`; see UnionAll Types.)

More examples will be discussed later in 메서드.

How does one construct a `Point` object? It is possible to define custom constructors for composite types, which will be discussed in detail in Constructors, but in the absence of any special constructor declarations, there are two default ways of creating new composite objects, one in which the type parameters are explicitly given and the other in which they are implied by the arguments to the object constructor.

Since the type `Point{Float64}` is a concrete type equivalent to `Point` declared with `Float64` in place of `T`, it can be applied as a constructor accordingly:

```
julia> Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}
```

For the default constructor, exactly one argument must be supplied for each field:

```
julia> Point{Float64}(1.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64)
[...]

julia> Point{Float64}(1.0,2.0,3.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64, ::Float64, ::Float64)
[...]
```

Only one default constructor is generated for parametric types, since overriding it is not possible. This constructor accepts any arguments and converts them to the field types.

In many cases, it is redundant to provide the type of `Point` object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply `Point` itself as a constructor, provided that the implied value of the parameter type `T` is unambiguous:

```
julia> Point(1.0,2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}

julia> Point(1,2)
Point{Int64}(1, 2)
```

```
julia> typeof(ans)
Point{Int64}
```

In the case of **Point**, the type of **T** is unambiguously implied if and only if the two arguments to **Point** have the same type. When this isn't the case, the constructor will fail with a MethodError:

```
julia> Point(1,2.5)
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, !Matched::T) where T at none:2
```

Constructor methods to appropriately handle such mixed cases can be defined, but that will not be discussed until later on in Constructors.

Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

```
julia> abstract type Pointy{T} end
```

With this declaration, **Pointy{T}** is a distinct abstract type for each type or integer value of **T**. As with parametric composite types, each such instance is a subtype of **Pointy**:

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

Parametric abstract types are invariant, much as parametric composite types are:

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

The notation `Pointy{<:Real}` can be used to express the Julia analogue of a covariant type, while `Pointy{>:Int}` the analogue of a contravariant type, but technically these represent sets of types (see UnionAll Types).

```
julia> Pointy{Float64} <: Pointy{<:Real}
true

julia> Pointy{Real} <: Pointy{>:Int}
true
```

Much as plain old abstract types serve to create a useful hierarchy of types over concrete types, parametric abstract types serve the same purpose with respect to parametric composite types. We could, for example, have declared `Point{T}` to be a subtype of `Pointy{T}` as follows:

```
julia> struct Point{T} <: Pointy{T}
           x::T
           y::T
       end
```

Given such a declaration, for each choice of `T`, we have `Point{T}` as a subtype of `Pointy{T}`:

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{AbstractString} <: Pointy{AbstractString}
true
```

This relationship is also invariant:

```
julia> Point{Float64} <: Pointy{Real}
false

julia> Point{Float64} <: Pointy{<:Real}
true
```

What purpose do parametric abstract types like `Pointy` serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line x = y:

```
julia> struct DiagPoint{T} <: Pointy{T}
           x::T
       end
```

Now both `Point{Float64}` and `DiagPoint{Float64}` are implementations of the `Pointy{Float64}` abstraction, and similarly for every other possible choice of type `T`. This allows programming to a common interface shared by all `Pointy` objects, implemented for both `Point` and `DiagPoint`. This cannot be fully demonstrated, however, until we have introduced methods and dispatch in the next section, 메서드.

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of `T` like so:

```
julia> abstract type Pointy{T<:Real} end
```

With such a declaration, it is acceptable to use any type that is a subtype of `Real` in place of `T`, but not types that are not subtypes of `Real`:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{AbstractString}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got Type{AbstractString}

julia> Pointy{1}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got Int64
```

Type parameters for parametric composite types can be restricted in the same manner:

```
struct Point{T<:Real} <: Pointy{T}
    x::T
    y::T
end
```

To give a real-world example of how all this parametric type machinery can be useful, here is the actual definition of Julia's `Rational` immutable type (except that we omit the constructor here for simplicity), representing an exact ratio of integers:

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
end
```

It only makes sense to take ratios of integer values, so the parameter type `T` is restricted to being a subtype of `Integer`, and a ratio of integers represents a value on the real number line, so any `Rational` is an instance of the `Real` abstraction.

Tuple Types

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. For example, a 2-element tuple type resembles the following immutable type:

```
struct Tuple2{A,B}
    a::A
    b::B
end
```

However, there are three key differences:

- Tuple types may have any number of parameters.
- Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are.
- Tuples do not have field names; fields are only accessed by index.

Tuple values are written with parentheses and commas. When a tuple is constructed, an appropriate tuple type is generated on demand:

```
julia> typeof((1,"foo",2.5))
Tuple{Int64,String,Float64}
```

Note the implications of covariance:

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false

julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

Intuitively, this corresponds to the type of a function's arguments being a subtype of the function's signature (when the signature matches).

Vararg Tuple Types

The last parameter of a tuple type can be the special type `Vararg`, which denotes any number of trailing elements:

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

Notice that `Vararg{T}` corresponds to zero or more elements of type `T`. Vararg tuple types are used to represent the arguments accepted by varargs methods (see 가변인자 함수).

The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `NTuple{N,T}` is a convenient alias for `Tuple{Vararg{T,N}}`, i.e. a tuple type containing exactly `N` elements of type `T`.

Named Tuple Types

Named tuples are instances of the `NamedTuple` type, which has two parameters: a tuple of symbols giving the field names, and a tuple type giving the field types.

```
julia> typeof((a=1,b="hello"))
NamedTuple{(:a, :b),Tuple{Int64,String}}
```

A `NamedTuple` type can be used as a constructor, accepting a single tuple argument. The constructed `NamedTuple` type can be either a concrete type, with both parameters specified, or a type that specifies only field names:

```
julia> NamedTuple{(:a, :b),Tuple{Float32, String}}((1,""))
(a = 1.0f0, b = "")

julia> NamedTuple{(:a, :b)}((1,""))
(a = 1, b = "")
```

If field types are specified, the arguments are converted. Otherwise the types of the arguments are used directly.

Singleton Types

There is a special kind of abstract parametric type that must be mentioned here: singleton types. For each type, `T`, the "singleton type" `Type{T}` is an abstract type whose only instance is the object `T`. Since the definition is a little difficult to parse, let's look at some examples:

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

In other words, `isa(A,Type{B})` is true if and only if `A` and `B` are the same object and that object is a type. Without the parameter, `Type` is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

```
julia> isa(Type{Float64}, Type)
true
```

```
julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true
```

Any object that is not a type is not an instance of `Type`:

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

Until we discuss Parametric Methods and conversions, it is difficult to explain the utility of the singleton type construct, but in short, it allows one to specialize function behavior on specific type values. This is useful for writing methods (especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

A few popular languages have singleton types, including Haskell, Scala and Ruby. In general usage, the term "singleton type" refers to a type whose only instance is a single value. This meaning applies to Julia's singleton types, but with that caveat that only type objects have singleton types.

Parametric Primitive Types

Primitive types can also be declared parametrically. For example, pointers are represented as primitive types which would be declared in Julia like this:

```
# 32-bit system:
primitive type Ptr{T} 32 end

# 64-bit system:
primitive type Ptr{T} 64 end
```

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter `T` is not used in the definition of the type itself – it is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus, `Ptr{Float64}` and `Ptr{Int64}` are distinct types, even though they have identical representations. And of course, all specific pointer types are subtypes of the umbrella `Ptr` type:

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

11.9 UnionAll Types

We have said that a parametric type like `Ptr` acts as a supertype of all its instances (`Ptr{Int64}` etc.). How does this work? `Ptr` itself cannot be a normal data type, since without knowing the type of the referenced data the type clearly cannot be used for memory operations. The answer is that `Ptr` (or other parametric types like `Array`) is a different kind of type called a `UnionAll` type. Such a type expresses the iterated union of types for all values of some parameter.

`UnionAll` types are usually written using the keyword `where`. For example `Ptr` could be more accurately written as `Ptr{T} where T`, meaning all values whose type is `Ptr{T}` for some value of `T`. In this context, the parameter `T` is also often called a "type variable" since it is like a variable that ranges over types. Each `where` introduces a single type variable, so these expressions are nested for types with multiple parameters, for example `Array{T,N} where N where T`.

The type application syntax `A{B,C}` requires `A` to be a `UnionAll` type, and first substitutes `B` for the outermost type variable in `A`. The result is expected to be another `UnionAll` type, into which `C` is then substituted. So `A{B,C}` is equivalent to `A{B}{C}`. This explains why it is possible to partially instantiate a type, as in `Array{Float64}`: the first parameter value has been fixed, but the second still ranges over all possible values. Using explicit `where` syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional arrays can be written as `Array{T,1} where T`.

Type variables can be restricted with subtype relations. `Array{T} where T<:Integer` refers to all arrays whose element type is some kind of `Integer`. The syntax `Array{<:Integer}` is a convenient shorthand for `Array{T} where T<:Integer`. Type variables can have both lower and upper bounds. `Array{T} where Int<:T<:Number` refers to all arrays of `Number`s that are able to contain `Int`s (since `T` must be at least as big as `Int`). The syntax `where T>:Int` also works to specify only the lower bound of a type variable, and `Array{>:Int}` is equivalent to `Array{T} where T>:Int`.

Since `where` expressions nest, type variable bounds can refer to outer type variables. For example `Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real` refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` of any kind of array whose element type contains the type of the first tuple element.

The `where` keyword itself can be nested inside a more complex declaration. For example, consider the two types created by the following declarations:

```
julia> const T1 = Array{Array{T,1} where T, 1}
Array{Array{T,1} where T,1}
```

```
julia> const T2 = Array{Array{T,1}, 1} where T
Array{Array{T,1},1} where T
```

Type `T1` defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. On the other hand, type `T2` defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Note that `T2` is an abstract type, e.g., `Array{Array{Int,1},1} <: T2`, whereas `T1` is a concrete type. As a consequence, `T1` can be constructed with a zero-argument constructor `a=T1()` but `T2` cannot.

There is a convenient syntax for naming such types, similar to the short form of function definition syntax:

```
Vector{T} = Array{T,1}
```

This is equivalent to `const Vector = Array{T,1} where T`. Writing `Vector{Float64}` is equivalent to writing `Array{Float64,1}`, and the umbrella type `Vector` has as instances all `Array` objects where the second parameter – the number of array dimensions – is 1, regardless of what the element type is. In languages where parametric types must always be specified in full, this is not especially helpful, but in Julia, this allows one to write just `Vector` for the abstract type including all one-dimensional dense arrays of any element type.

11.10 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. This can be done with a simple assignment statement. For example, `UInt` is aliased to either `UInt32` or `UInt64` as is appropriate for the size of pointers on the system:

```
# 32-bit system:
julia> UInt
UInt32

# 64-bit system:
julia> UInt
UInt64
```

This is accomplished via the following code in `base/boot.jl`:

```
if Int === Int64
    const UInt = UInt64
```

```
else
    const UInt = UInt32
end
```

Of course, this depends on what `Int` is aliased to – but that is predefined to be the correct type – either `Int32` or `Int64`.

(Note that unlike `Int`, `Float` does not exist as a type alias for a specific sized `AbstractFloat`. Unlike with integer registers, where the size of `Int` reflects the size of a native pointer on that machine, the floating point register sizes are specified by the IEEE-754 standard.)

11.11 Operations on Types

Since types in Julia are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the `<:` operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The `isa` function tests if an object is of a given type and returns true or false:

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

The `typeof` function, already used throughout the manual in examples, returns the type of its argument. Since, as noted above, types are objects, they also have types, and we can ask what their types are:

```
julia> typeof(Rational{Int})
DataType

julia> typeof(Union{Real,String})
Union
```

What if we repeat the process? What is the type of a type of a type? As it happens, types are all composite values and thus all have a type of `DataType`:

```
julia> typeof(DataType)
DataType
```

```
julia> typeof(Union)
DataType
```

`DataType` is its own type.

Another operation that applies to some types is `supertype`, which reveals a type's supertype. Only declared types (`DataType`) have unambiguous supertypes:

```
julia> supertype(Float64)
AbstractFloat

julia> supertype(Number)
Any

julia> supertype(AbstractString)
Any

julia> supertype(Any)
Any
```

If you apply `supertype` to other type objects (or non-type objects), a `MethodError` is raised:

```
julia> supertype(Union{Float64,Int64})
ERROR: MethodError: no method matching supertype(::Type{Union{Float64, Int64}})
Closest candidates are:
  supertype(!Matched::DataType) at operators.jl:42
  supertype(!Matched::UnionAll) at operators.jl:47
```

11.12 Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the `show` function. For example, suppose we define a type to represent complex numbers in polar form:

```
julia> struct Polar{T<:Real} <: Number
           r::T
           θ::T
       end

julia> Polar(r::Real,θ::Real) = Polar(promote(r,θ)...)
Polar
```

Here, we've added a custom constructor function so that it can take arguments of different `Real` types and promote them to a common type (see Constructors and Conversion and Promotion). (Of course, we would have to define lots of other methods, too, to make it act like a `Number`, e.g. `+`, `*`, `one`, `zero`, promotion rules and so on.) By default, instances of this type display rather simply, with information about the type name and the field values, as e.g. `Polar{Float64}(3.0,4.0)`.

If we want it to display instead as `3.0 * exp(4.0im)`, we would define the following method to print the object to a given output object `io` (representing a file, terminal, buffer, etcetera; see Networking and Streams):

```
julia> Base.show(io::IO, z::Polar) = print(io, z.r, " * exp(", z.θ, "im)")
```

More fine-grained control over display of `Polar` objects is possible. In particular, sometimes one wants both a verbose multi-line printing format, used for displaying a single object in the REPL and other interactive environments, and also a more compact single-line format used for `print` or for displaying the object as part of another object (e.g. in an array). Although by default the `show(io, z)` function is called in both cases, you can define a different multi-line format for displaying an object by overloading a three-argument form of `show` that takes the `text/plain` MIME type as its second argument (see Multimedia I/O), for example:

```
julia> Base.show(io::IO, ::MIME"text/plain", z::Polar{T}) where{T} =
           print(io, "Polar{$T} complex number:\n   ", z)
```

(Note that `print(..., z)` here will call the 2-argument `show(io, z)` method.) This results in:

```
julia> Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> [Polar(3, 4.0), Polar(4.0,5.3)]
2-element Array{Polar{Float64},1}:
 3.0 * exp(4.0im)
 4.0 * exp(5.3im)
```

where the single-line `show(io, z)` form is still used for an array of `Polar` values. Technically, the REPL calls `display(z)` to display the result of executing a line, which defaults to `show(stdout, MIME("text/plain"), z)`, which in turn defaults to `show(stdout, z)`, but you should not define new `display` methods unless you are defining a new multimedia display handler (see Multimedia I/O).

Moreover, you can also define `show` methods for other MIME types in order to enable richer display (HTML, images, etcetera) of objects in environments that support this (e.g. IJulia). For example, we can define formatted HTML display of `Polar` objects, with superscripts and italics, via:

```
julia> Base.show(io::IO, ::MIME"text/html", z::Polar{T}) where {T} =
           println(io, "<code>Polar{$T}</code> complex number: ",
                   z.r, " <i>e</i><sup>", z.θ, " <i>i</i></sup>")
```

A `Polar` object will then display automatically using HTML in an environment that supports HTML display, but you can call `show` manually to get HTML output if you want:

```
julia> show(stdout, "text/html", Polar(3.0,4.0))
<code>Polar{Float64}</code> complex number: 3.0 <i>e</i><sup>4.0 <i>i</i></sup>
```

As a rule of thumb, the single-line `show` method should print a valid Julia expression for creating the shown object. When this `show` method contains infix operators, such as the multiplication operator (`*`) in our single-line `show` method for `Polar` above, it may not parse correctly when printed as part of another object. To see this, consider the expression object (see Program representation) which takes the square of a specific instance of our `Polar` type:

```
julia> a = Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> print(:($a^2))
3.0 * exp(4.0im) ^ 2
```

Because the operator `^` has higher precedence than `*` (see Operator Precedence and Associativity), this output does not faithfully represent the expression `a ^ 2` which should be equal to `(3.0 * exp(4.0im)) ^ 2`. To solve this issue, we must make a custom method for `Base.show_unquoted(io::IO, z::Polar, indent::Int, precedence::Int)`, which is called internally by the expression object when printing:

```
julia> function Base.show_unquoted(io::IO, z::Polar, ::Int, precedence::Int)
           if Base.operator_precedence(:*) <= precedence
               print(io, "(")
               show(io, z)
               print(io, ")")
           else
               show(io, z)
           end
       end

julia> :($a^2)
:((3.0 * exp(4.0im)) ^ 2)
```

The method defined above adds parentheses around the call to `show` when the precedence of the calling operator is higher than or equal to the precedence of multiplication. This check allows expressions which parse correctly without the parentheses (such as `:($a + 2)` and `:($a == 2)`) to omit them when printing:

```
julia> :($a + 2)
:(3.0 * exp(4.0im) + 2)

julia> :($a == 2)
:(3.0 * exp(4.0im) == 2)
```

In some cases, it is useful to adjust the behavior of `show` methods depending on the context. This can be achieved via the `IOContext` type, which allows passing contextual properties together with a wrapped IO stream. For example, we can build a shorter representation in our `show` method when the `:compact` property is set to `true`, falling back to the long representation if the property is `false` or absent:

```
julia> function Base.show(io::IO, z::Polar)
           if get(io, :compact, false)
               print(io, z.r, " ", z.θ, "im")
           else
               print(io, z.r, " * exp(", z.θ, "im)")
           end
       end
```

This new compact representation will be used when the passed IO stream is an `IOContext` object with the `:compact` property set. In particular, this is the case when printing arrays with multiple columns (where horizontal space is limited):

```
julia> show(IOContext(stdout, :compact=>true), Polar(3, 4.0))
3.0 4.0im

julia> [Polar(3, 4.0) Polar(4.0,5.3)]
1×2 Array{Polar{Float64},2}:
 3.0 4.0im  4.0 5.3im
```

See the `IOContext` documentation for a list of common properties which can be used to adjust printing.

11.13 "Value types"

In Julia, you can't dispatch on a value such as `true` or `false`. However, you can dispatch on parametric types, and Julia allows you to include "plain bits" values (Types, Symbols, Integers, floating-point numbers, tuples, etc.) as type parameters. A common example is the dimensionality parameter in `Array{T,N}`, where `T` is a type (e.g., `Float64`) but `N` is just an `Int`.

You can create your own custom types that take values as parameters, and use them to control dispatch of custom types. By way of illustration of this idea, let's introduce a parametric type, `Val{x}`, and a constructor `Val(x) = Val{x}()`, which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

`Val` is defined as:

```
julia> struct Val{x}
       end

julia> Val(x) = Val{x}()
Val
```

There is no more to the implementation of `Val` than this. Some functions in Julia's standard library accept `Val` instances as arguments, and you can also use it to write your own functions. For example:

```
julia> firstlast(::Val{true}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Val{false}) = "Last"
firstlast (generic function with 2 methods)

julia> firstlast(Val(true))
"First"

julia> firstlast(Val(false))
"Last"
```

For consistency across Julia, the call site should always pass a `Val`instance rather than using a type, i.e., use `foo(Val(:bar))` rather than `foo(Val{:bar})`.

It's worth noting that it's extremely easy to mis-use parametric "value" types, including `Val`; in unfavorable cases, you can easily end up making the performance of your code much worse. In particular, you would never want to

write actual code as illustrated above. For more information about the proper (and improper) uses of `Val`, please read the more extensive discussion in the performance tips.

[1] "Small" is defined by the `MAX_UNION_SPLITTING` constant, which is currently set to 4.

Chapter 12

메서드

함수에서 함수는 인수의 튜플을 반환 값으로 매핑하는 객체이거나 적절한 값을 반환 할 수없는 경우 예외를 throw합니다. 두 가지 정수를 더하는 것은 부동 소수점 수에 정수를 더하는 것과는 다른 두 개의 부동 소수점 수를 더하는 것과는 매우 다릅니다. 이러한 구현의 차이점에도 불구하고 이러한 작업은 모두 "추가"라는 일반적인 개념에 속합니다. 따라서 줄리아에서 이러한 동작은 모두 하나의 객체인 + 함수에 속합니다.

동일한 개념의 여러 구현을 원활하게 사용하기 위해 함수를 한꺼번에 정의할 필요는 없지만 인수 유형 및 개수의 특정 조합에 특정 동작을 제공함으로써 구분할 수 있습니다. 함수에 대해 가능한 한 행동의 정의를 메서드 라고합니다. 지금까지 우리는 하나의 메서드로 정의 된 함수의 예제만을 제시했으며, 모든 유형의 인수에 적용 할 수 있습니다. 그러나 메서드 정의의 서명에는 그 수 이외에 인수의 유형을 표시하기 위해 주석을 달 수 있으며 단일 메서드 정의 이상이 제공 될 수 있습니다. 함수가 특정 튜플의 인수에 적용되면 해당 인수에 적용 할 수있는 가장 구체적인 메서드가 적용됩니다. 따라서 함수의 전반적인 동작은 다양한 메서드 정의의 동작을 패치하는 것입니다. 패치 워크가 잘 설계된 경우, 메서드의 구현이 상당히 다를 수도 있지만, 함수의 외부 동작은 매끄럽고 일관성있게 나타납니다.

함수가 적용될 때 실행할 메서드의 선택은 dispatch 라고 합니다. 줄리아는 파견 프로세스가 주어진 인수의 수와 함수의 모든 인수 유형에 따라 함수의 메서드 중 어느 것을 호출 할 것인지 선택할 수 있습니다. 이것은 전통적인 객체 지향 언어와는 다르다. 디스패치는 특별한 인수 구문을 사용하는 첫 번째 인수에만 기반을 두며 종종 인수로 명시 적으로 작성되지 않고 암시된다.[1] 함수의 인수를 모두 사용하여 첫 번째 메서드가 아닌 호출 할 메서드를 선택하는 것이 다중 디스패치로 알려져 있습니다. 다중 디스패치는 수학적 코드에서 특히 유용합니다. 다른 연산보다 하나의 인수에 "속하는" 연산을 인위적으로 판단하는 것은 의미가 없습니다. `x + y` 의 더하기 연산이 `y` 보다 `x` 에 속해 있습니까? 수학 연산자의 구현은 일반적으로 모든 인수의 유형에 따라 다릅니다. 그러나 수학적 작업을 넘어서더라도, 다중 파견은 프로그램을 구조화하고 조직화하는 데있어 유쾌하고 편리한 패러다임입니다.

[1] 예를 들어,`obj.meth(arg1, arg2)` 와 같은 메서드 호출에서 C ++이나 Java에서 객체 obj는 메서드 호출을 "받는다"는 의미가 아니라 `this` 키워드를 통해 메서드에 암묵적으로 전달됩니다. 명시적인 메서드 인수. 현재 `this` 객체가 메서드 호출의 수신자 일 때 `meth (arg1, arg2)` 만 쓰고 `this` 를 수신 객체로 함축하여 생략 할 수 있습니다.

12.1 메서드 정의

지금까지 우리는 예제에서 제약되지 않은 인자 타입을 가진 하나의 메서드를 가진 함수만을 정의했다. 이러한 함수는 전통적인 동적 유형 지정 언어 에서처럼 작동합니다. 그럼에도 불구하고 우리는 다중 디스패치와 메서드를 의식하지 않고 거의 연속적으로 사용했습니다. 앞서 말한 + 함수와 같은 모든 줄리아의 표준 함수와 연산자에는 인수 유형과 개수의 다양한 조합을 통해 동작을 정의하는 많은 메서드가 있습니다.

함수를 정의 할 때 Composite Types 섹션에서 소개 한 :: type-assertion 연산자를 사용하여 적용 할 수있는 매개 변수의 유형을 선택적으로 제한 할 수 있습니다:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

이 함수 정의는 x 와 y 가 모두 타입의 값인 호출에만 적용됩니다 Float64:

```
julia> f(2.0, 3.0)
7.0
```

다른 유형의 인수에 적용하면 MethodError 가됩니다.

```
julia> f(2.0, 3)
ERROR: MethodError: no method matching f(::Float64, ::Int64)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)
Closest candidates are:
  f(!Matched::Float64, ::Float64) at none:1

julia> f(2.0, "3.0")
ERROR: MethodError: no method matching f(::Float64, ::String)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f("2.0", "3.0")
ERROR: MethodError: no method matching f(::String, ::String)
```

보시다시피, 인수는 정확히 Float64 유형이어야합니다. 정수 또는 32 비트 부동 소수점 값과 같은 다른 숫자 유형은 자동으로 '64 비트 부동 소수점으로 변환되지 않으며 숫자로 해석되는 문자열도 아닙니다. Float64 는 구체적인 타입이고 줄리아에서

구체적인 타입을 서브 클래 싱 할 수 없기 때문에, 그러한 정의는 정확히 Float64 타입의 인자에만 적용될 수 있습니다. 그러나 선언 된 매개 변수 유형이 추상 인 경우보다 일반적인 메서드를 작성하는 것이 종종 유용 할 수 있습니다.

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

이 메서드 정의는 Number 의 인스턴스 인 모든 인수 쌍에 적용됩니다. 그들은 각각 숫자 값인 한 동일한 유형일 필요는 없습니다. 서로 다른 숫자 타입을 처리하는 문제는 2x - y 표현식의 산술 연산에 위임 됩니다.

여러 메서드로 함수를 정의하려면 함수의 수와 유형을 여러 번 정의해야합니다. 함수에 대한 첫 번째 메서드 정의는 함수 개체를 만들고 후속 메서드 정의는 기존 함수 개체에 새 메서드를 추가합니다. 인수의 수와 유형과 일치하는 가장 구체적인 메서드 정의는 함수가 적용될 때 실행됩니다. 따라서, 위의 두 메서드 정의는 함께 추상 타입 Number 의 모든 인스턴스 쌍에 대해 f 에 대한 동작을 정의하지만 Float64 쌍에 고유 한 다른 동작을 사용하여 정의됩니다. 값. 인수 중 하나가 64 비트 부동 소수점이지만 다른 하나는 부동 소수점인 경우 f(Float64, Float64) 메서드를 호출 할 수 없으며 보다 일반적인 f(Number, Number) 메서드를 사용해야합니다.

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

2x + y 정의는 첫 번째 경우에만 사용되는 반면, 2x-y 정의는 다른 것에 사용됩니다. 자동 주조 또는 함수 인수 변환이 수행되지 않습니다. 줄리아의 모든 변환은 비 마법적이고 완전히 명시적입니다. 그러나 전환 및 판촉은 충분히 발전된 기술의 영리한 적용이 마법과 구별 될 수 없음을 보여줍니다. [2]

숫자가 아닌 값과 2 개보다 적거나 많은 인수의 경우, f 함수는 정의되지 않은 채로 남아 있으며, 여전히 MethodError 가 됩니다:

```
julia> f("foo", 3)
ERROR: MethodError: no method matching f(::String, ::Int64)
```

```
Closest candidates are:
  f(!Matched::Number, ::Number) at none:1

julia> f()
ERROR: MethodError: no method matching f()
Closest candidates are:
  f(!Matched::Float64, !Matched::Float64) at none:1
  f(!Matched::Number, !Matched::Number) at none:1
```

대화형 세션에서 함수 객체 자체를 입력하면 함수에 어떤 메서드가 있는지 쉽게 알 수 있습니다.

```
julia> f
f (generic function with 2 methods)
```

이 출력은 `f` 가 두 가지 메서드를 가진 함수 객체라는 것을 알려줍니다. 이러한 메서드의 서명이 무엇인지 알아 보려면 `methods` 함수를 사용하십시오.

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x::Float64, y::Float64) in Main at none:1
[2] f(x::Number, y::Number) in Main at none:1
```

`f` 에는 두 개의 Float64 인수를 취하는 메서드와 `Number` 타입의 인수를 취하는 메서드가 있습니다. 또한 메서드가 정의 된 파일과 행 번호를 표시합니다.이 메서드가 REPL에 정의 되었기 때문에 명백한 행 번호는 `none:1` 입니다.

`::` 를 사용한 타입 선언이 없으면 메서드 매개 변수의 타입은 기본적으로 `Any` 이며, 이는 줄리아의 모든 값이 추상 타입 `Any` 의 인스턴스이기 때문에 제약이 없다는 것을 의미합니다. 따라서 `f` 에 대한 catch-all 메서드를 다음과 같이 정의 할 수 있습니다 :

```
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> f("foo", 1)
Whoa there, Nelly.
```

이 catch-all은 한 쌍의 매개 변수 값에 대해 가능한 다른 메서드 정의보다 덜 구체적이므로 다른 메서드 정의가 적용되지 않는 인수 쌍에서만 호출됩니다.

단순한 개념으로 보일지라도, 가치 유형에 대한 다중 파견은 아마도 줄리아 언어의 가장 강력하고 핵심적인 단일 기능 일 것입니다. 핵심 운영에는 일반적으로 수십 가지 방법이 있습니다:

```
julia> methods(+)
# 180 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:227
[2] +(x::Bool, y::Bool) in Base at bool.jl:89
[3] +(x::Bool) in Base at bool.jl:86
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
[5] +(x::Bool, z::Complex) in Base at complex.jl:234
[6] +(a::Float16, b::Float16) in Base at float.jl:373
[7] +(x::Float32, y::Float32) in Base at float.jl:375
[8] +(x::Float64, y::Float64) in Base at float.jl:376
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:228
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:242
[11] +(x::Char, y::Integer) in Base at char.jl:40
[12] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:307
[13] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:392
[14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:391
[15] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:390
[16] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:361
[17] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:398
...
[180] +(a, b, c, xs...) in Base at operators.jl:424
```

유연한 파라 메트릭 유형 시스템과의 다중 디스패치 기능을 통해 줄리아는 구현 세부 정보에서 분리 된 상위 수준의 알고리즘을 추상적으로 표현할 수 있지만 런타임에 각 사례를 처리 할 수있는 효율적인 특수 코드를 생성 할 수 있습니다.

12.2 방법 모호성

일부 인수 조합에 적용 할 수있는 고유 한 가장 구체적인 메서드가 없도록 함수 메서드 세트를 정의 할 수 있습니다:

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
```

```
8.0

julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous. Candidates:
  g(x, y::Float64) in Main at none:1
  g(x::Float64, y) in Main at none:1
Possible fix, define
  g(::Float64, ::Float64)
```

여기서 `g (2.0, 3.0)` 호출은 `g (Float64, Any)` 또는 `g (Any, Float64)` 메서드에 의해 처리 될 수 있으며, 어느 쪽도 더 구체적이지 않습니다. 이런 경우 줄리아는 임의로 메서드를 선택하는 것이 아니라 `MethodError` 를 발생시킵니다. 교차 사례의 적절한 방법을 지정하여 메서드 모호성을 피할 수 있습니다:

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

일시적인 경우보다 구체적인 방법이 정의 될 때까지 모호성이 존재하기 때문에 모호성을 제거하는 방법을 먼저 정의하는 것이 좋습니다.

보다 복잡한 경우, 방법 모호성 해결 은 디자인의 특정 요소를 포함합니다. 이 주제는 아래 에서 더 자세히 다루어 집니다.

12.3 파라메트릭 메서드

메서드 정의는 선택적으로 서명을 한정하는 매개 변수를 가질 수 있습니다.

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

첫 번째 방법은 두 인수가 모두 동일한 구체 유형일 때마다 적용됩니다. 두 번째 방법은 다른 모든 경우를 포괄하는 포괄적인 방식으로 작동합니다. 따라서 전반적으로 두 인수가 같은 유형인지 여부를 확인하는 부울 함수를 정의합니다.

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(Int32(1), Int64(2))
false
```

이러한 정의는 형식 서명이 `UnionAll` (UnionAll Types 참조) 유형인 메서드에 해당합니다.

이러한 종류의 파견에 의한 기능 행동의 정의는 매우 흔하며, 심지어 줄리아에서도 관용적입니다. 메서드 유형 매개 변수는 인수의 유형으로 사용되는 것으로 제한되지 않으며 함수의 본문 또는 본문에 값이있는 모든 위치에서 사용할 수 있습니다. 다음은 메서드 시그니처의 매개 변수 유형 `Vector {T}` 에 대한 유형 매개 변수로 메서드 유형 매개 변수 `T` 가 사용되는 예제입니다.

```
julia> myappend(v::Vector{T}, x::T) where {T} = [v..., x]
myappend (generic function with 1 method)

julia> myappend([1,2,3],4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError: no method matching myappend(::Array{Int64,1}, ::Float64)
Closest candidates are:
```

```
  myappend(::Array{T,1}, !Matched::T) where T at none:1

julia> myappend([1.0,2.0,3.0],4.0)
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Array{Float64,1}, ::Int64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1
```

보시다시피, 첨부된 요소의 유형은 추가되는 벡터의 요소 유형과 일치해야합니다. 그렇지 않으면 `MethodError` 가 발생합니다. 다음 예제에서는 메서드 유형 매개 변수 `T` 가 반환 값으로 사용됩니다.

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

타입 선언에 타입 파라미터에 하위 타입 제약 조건을 넣을 수있는 것처럼 (파라메트릭 타입 참조), 메서드의 타입 파라미터를 제한 할 수도 있습니다:

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (generic function with 1 method)

julia> same_type_numeric(x::Number, y::Number) = false
same_type_numeric (generic function with 2 methods)

julia> same_type_numeric(1, 2)
true

julia> same_type_numeric(1, 2.0)
```

```
false

julia> same_type_numeric(1.0, 2.0)
true

julia> same_type_numeric("foo", 2.0)
ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)
Closest candidates are:
  same_type_numeric(!Matched::T, ::T) where T<:Number at none:1
  same_type_numeric(!Matched::Number, ::Number) at none:1

julia> same_type_numeric("foo", "bar")
ERROR: MethodError: no method matching same_type_numeric(::String, ::String)

julia> same_type_numeric(Int32(1), Int64(2))
false
```

`same_type_numeric` 함수는 위에 정의된 `same_type` 함수와 매우 비슷하게 동작하지만 숫자 쌍에 대해서만 정의됩니다.

파라 메트릭 메서드는 타입 작성에 사용되는 `where` 표현식과 같은 구문을 허용합니다 (UnionAll Types 참조). 하나의 매개 변수 만있는 경우에는 `{T}` 에있는 중괄호를 생략 할 수 있지만 명확성을 위해 선호하는 경우가 많습니다. 여러 매개 변수는 쉼표로 구분할 수 있습니다. e.g. `where {T, S<:Real}`, 또는 `where` 중첩을 사용하여 작성 , e.g. `where S<:Real where T`.

12.4 메서드 재정의

메서드를 재정의하거나 새 메서드를 추가 할 때 이러한 변경 사항이 즉시 적용되지 않는다는 것을 인식하는 것이 중요합니다. 줄리아가 일반적인 JIT 트릭이나 오버 헤드없이 정적으로 코드를 추론하고 컴파일하여 실행하는 것이 중요합니다. 실제로 새로운 메서드 정의는 Tasks와 쓰레드 (그리고 이전에 정의 된 `@generated` 함수들) 를 포함하여 현재 런타임 환경에서 볼 수 없을 것입니다. 예를 들어 이것이 무엇을 의미하는지 알아 봅시다.

```
julia> function tryeval()
           @eval newfun() = 1
           newfun()
       end
tryeval (generic function with 1 method)

julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx1, while current world is xxxx2.
```

```
Closest candidates are:
  newfun() at none:1 (method too new to be called from this world context.)
 in tryeval() at none:1
 ...

julia> newfun()
1
```

이 예제에서는 newfun 에 대한 새로운 정의가 생성되었지만 즉시 호출 할 수 없음을 관찰합니다. 새로운 전역 변수는 tryeval 함수에서 즉시 볼 수 있으므로 return newfun (괄호없이)을 쓸 수 있습니다. 그러나 당신이나 당신의 호출자, 또는 그들이 부르는 기능도 아닙니다. 이 새로운 메서드 정의를 호출 할 수 있습니다!

그러나 예외가 있습니다 : 미래의newfun 호출은 예상대로 REPL 에서 작동합니다. newfun의 새로운 정의를보고 호출 할 수 있습니다.

그러나 'tryeval'에 대한 향후 호출은 REPL에서 처럼 newfun 의 정의를 계속 볼 것이며, 따라서 tryeval 에 대한 호출 전에 나타납니다.

어떻게 작동하는지 직접 확인해보십시오.

이 동작의 구현은 "세계 시대 카운터"입니다.이 단조 증가 값은 각 메서드 정의 연산을 추적합니다. 이를 통해 "주어진 런타임 환경에서 볼 수있는 메서드 정의 세트"를 단일 숫자 또는 "world age"로 설명 할 수 있습니다. 또한 서수 값을 비교하여 두 worlds에서 사용할 수있는 방법을 비교할 수 있습니다. 위 예제에서 "current world" ("newfun" 메서드가 있는)는 tryeval 실행이 시작될 때 수정 된 작업 로컬 "런타임 환경" 보다 큰 것입니다.

때로는 이것을 피하는 것이 필요합니다 (예를 들어 위의 REPL을 구현하는 경우). 다행히도 쉬운 해결책이 있습니다 : Base.invokelatest를 사용하여 함수를 호출하십시오 :

```
julia> function tryeval2()
           @eval newfun2() = 2
           Base.invokelatest(newfun2)
       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

마지막으로, 이 규칙이 적용되는 좀 더 복잡한 예제를 살펴 보겠습니다. 처음에는 하나의 메서드를 가지고있는 함수 f(x) 를 정의합니다 :

```
julia> f(x) = "기본 정의"
f (generic function with 1 method)
```

f (x)를 사용하는 다른 연산을 시작합니다.:

```
julia> g(x) = f(x)
g (generic function with 1 method)

julia> t = @async f(wait()); yield();
```

이제 우리는f (x)에 몇 가지 새로운 메서드를 추가합니다 :

```
julia> f(x::Int) = "Int로 정의"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "Type{Int}로 정의"
f (generic function with 3 methods)
```

결과가 어떻게 다른지 비교해봅시다.:

```
julia> f(1)
"Int로 정의"

julia> g(1)
"Int로 정의"

julia> fetch(schedule(t, 1))
"기본 정의"

julia> t = @async f(wait()); yield();

julia> fetch(schedule(t, 1))
"Int로 정의"
```

12.5 파라 메트릭 방법을 사용한 디자인 패턴

성능이나 유용성에 복잡한 디스패치 로직이 필요하지는 않지만 때로는 일부 알고리즘을 표현하는 가장 좋은 방법 일 수 있습니다. 이런 방식으로 디스패치를 사용할 때 때로는 나타나는 몇 가지 일반적인 디자인 패턴이 있습니다.

super-type 에서 type 매개 변수 추출

여기 `AbstractArray` 의 임의의 하위 유형의 요소 유형 `T` 를 반환하기위한 올바른 코드 템플릿이 있습니다.:

```
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

삼각 파견을 사용합니다. 예를 들어 `T` 가 `UnionAll` 타입 인 경우에 주목합니다. `eltype(Array{T} where T <: Integer)` 이면 `Any` 가 반환됩니다. (`Base` 에있는 `eltype` 의 버전도 마찬가지입니다).

줄리아 v0.6에서 삼각형 파견의 출현 이전에 유일한 올바른 방법이었던 또 다른 방법은 다음과 같습니다:

```
abstract type AbstractArray{T, N} end
eltype(::Type{AbstractArray}) = Any
eltype(::Type{AbstractArray{T}}) where {T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A<:AbstractArray} = eltype(supertype(A))
```

또다른 가능성은 다음과 같습니다. 매개 변수 `T` 가 더 좁게 일치해야하는 경우에 적용하는 것이 유용 할 수 있습니다:

```
eltype(::Type{AbstractArray{T, N} where {T<:S, N<:M}}) where {M, S} = Any
eltype(::Type{AbstractArray{T, N} where {T<:S}}) where {N, S} = Any
eltype(::Type{AbstractArray{T, N} where {N<:M}}) where {M, T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A <: AbstractArray} = eltype(supertype(A))
```

일반적인 실수 중 하나는 내부 검사를 사용하여 요소 유형을 얻는 것입니다. :

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

그러나 이것이 실패할 경우를 만드는 것은 어렵지 않습니다:

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

여기서 우리는 매개 변수를 가지지 않는 `BitVector` 타입을 만들었지만, element-type이 여전히 완전하게 지정되고 `T` 는 `Bool` 과 같습니다!

다른 형식 매개 변수를 사용하여 비슷한 유형 만들기

일반적인 코드를 만들 때 유형의 레이아웃을 변경하고 유사한 유형의 객체를 생성해야하는 경우가 종종 있습니다. 유형 매개 변수의 변경이 필요합니다. 예를 들어 임의의 요소 유형을 가진 일종의 추상 배열을 가질 수 있으며 특정 요소 유형으로 계산을 작성하려고합니다. 이 유형 변환을 계산하는 방법을 설명하는 각 `AbstractArray{T}` 하위 유형에 대한 메서드를 구현해야합니다. 하나의 부속 유형에 대해 다른 매개 변수를 갖는 다른 부속 유형으로의 변환이 없습니다. (빠른 검토 : 이것이 왜 있는지 보시겠습니까?)

`AbstractArray` 의 부속유형은 일반적으로이를 달성하는 두가지 메서드를 구현합니다. 입력배열을 특정 `AbstractArray {T, N}` 추상유형의 부속유형으로 변환하는 메서드 및 특정요소 유형을 가진 초기화되지 않은 새로운 배열을 만드는 방법. 이들의 샘플구현은 줄리아 Base에서 찾을 수 있습니다. 다음은 `input` 과 `output` 이 같은 타입으로되어 있다는 것을 보장하는 기본적인 예제사용법입니다 :

```
input = convert(AbstractArray{Eltype}, input)
output = similar(input, Eltype)
```

이를 확장하기 위해 알고리즘에 입력배열의 복사본이 필요한경우 반환값이 원래입력의 별칭일 수 있으므로 `convert` 는 충분하지 않습니다. 출력배열을 만들기 위해 `similar`와 입력데이터로 채우기 위해 `copyto!` 를 결합하는 것은 변경가능한 입력인수 복사본에 대한 요구사항을 표현하는 일반적인 방법입니다.:

```
copy_with_eltype(input, Eltype) = copyto!(similar(input, Eltype), input)
```

반복 디스패치

다단계 매개변수 목록을 발송하려면 종종 각 발송 단계를 별개의 기능으로 분리하는 것이 가장 좋습니다. 단일발송에 대한 접근방식과 비슷할 수도 있지만, 아래에서 보게 되겠지만 더 유연합니다.

예를들어 배열의 요소유형을 디스패칭하려고하면 종종 모호한 상황이 발생합니다. 대신 일반적으로 코드는 컨테이너유형에 먼저 전달되고 eltype를 기반으로 한 것 보다 구체적인 메서드로 순환됩니다. 대부분의 경우 알고리즘은이 계층적 접근방식을 편리하게 사용하지만 다른 경우에는 수동으로 이 엄격성을 해결해야합니다. 이 디스패칭 브랜칭은 두 개의 행렬을 합산하는 로직에서 볼 수 있습니다. :

```
# 첫 번째 디스패치는 요소별 합계에 대한 map 알고리즘을 선택합니다.
+(a::Matrix, b::Matrix) = map(+, a, b)
# 그런 다음 디스패치는 각 요소를 처리하고 적절한 요소를 선택합니다.
# 계산을위한 공통 요소 유형.
+(a, b) = +(promote(a, b)...)
# 요소의 유형이 같으면 추가 할 수 있습니다.
# 예를 들어, 프로세서에 의해 노출된 원시연산을 통해.
+(a::Float64, b::Float64) = Core.add(a, b)
```

Trait-based 디스패치

위의 iterated 디스패치를 자연스럽게 확장하면 형식계층에 정의된 집합과 독립적인 형식집합을 디스패치 할 수있는 계층을 메서드선택에 추가 할 수 있습니다. 우리는 문제의유형의 **조합** 을 작성하여 그러한 집합을 구성 할 수 있지만, 생성 후에 **연합** 유형을 변경할 수 없으므로이 집합은 확장 할 수 없습니다. 그러나 이러한 확장가능 세트는 종종 "Holy-trait" 라는 디자인 패턴으로 프로그래밍 될 수 있습니다.

이 패턴은 함수인수가 속할 수있는 각 특성집합에 대해 다른 단일값 (또는 유형)을 계산하는 일반함수를 정의하여 구현됩니다. 이 기능이 순수하면 정상적인 발송과 비교하여 성능에 영향을주지 않습니다.

이전 섹션의 예제는 `map` 및 `promote`의 구현세부사항을 설명했습니다.이 두 특성은 이러한 특성에 따라 작동합니다. `map` 의 구현과 같이 행렬을 반복 할 때 중요한 질문 중 하나는 데이터를 순회하기 위해 사용하는 순서입니다. `AbstractArray` 보조형이 `Base.IndexStyle` 형질을 구현할 때 `map` 과 같은 다른 함수는 이러한 정보를 보내서 최상의 알고리즘을 선택합니다 (Abstract Array Interface). 즉, 일반정의 + 특성클래스를 사용하면 시스템에서 가장 빠른 버전을 선택할 수 있기 때문에 각 하위유형에서 `map` 의 사용자정의버전을 구현할 필요가 없습니다. trait-based의 디스패치를 보여주는 `map` 의 장난감 구현:

```
map(f, a::AbstractArray, b::AbstractArray) = map(Base.IndexStyle(a, b), f, a, b)
# 일반적인 구현 :
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray) = ...
# 선형 인덱싱 구현 (더 빠름)
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) = ...
```

이 trait-based 접근법은 스칼라 `+` 에 의해 채택 된 `promote` 메커니즘에도 존재합니다. 이것은 `promote_type` 을 사용하는데, 이것은 최적의 공통 유형을 반환합니다. 두 가지 유형의 피연산자가 주어진 경우 연산을 계산합시다. 이를 통해 모든 유형의 인수에 대해 모든 함수를 구현하는 문제를 줄이고, 각 유형에서 일반 유형으로 변환 작업을 구현하는 훨씬 작은 문제와 선호되는 preferred pair-wise promotion rules표를 줄일 수 있습니다.

출력 유형 계산

trait-based 프로모션에 대한 논의는 다음 디자인패턴으로의 전환을 제공합니다. 매트릭스 작동을위한 출력 요소 유형 계산.

추가 같은 기본 연산을 구현하기 위해 `promote_type` 함수를 사용하여 원하는 출력유형을 계산합니다. (이전과 마찬가지로 `+` 호출로 `promote` 호출에서 이것을 보았습니다).

행렬에 대한 함수의 경우보다 더 복잡한 연산 순서에 대해 예상되는 반환 형식을 계산해야 할 수도 있습니다. 이 작업은 종종 다음 단계로 수행됩니다.

1. 알고리즘의 커널에 의해 수행되는 일련의 연산을 나타내는 작은 함수 `op` 를 작성합니다.
2. 결과 행렬의 요소 타입 `R` 을`promote_op (op, argument_types ...)` 로 계산합니다. 여기서 `argument_types` 는 각 입력 배열에 적용된 `eltype` 에서 계산됩니다.

3. 출력 행렬을 `similar(R, dims)` 로 만듭니다. 여기서 `dims` 는 출력 배열의 원하는 차원입니다.

For a more specific example, a generic square-matrix multiply pseudo-code might look like:

좀 더 구체적인 예를 들어, 일반제곱 - 행렬곱셈 pseudo코드는 다음과 같습니다.

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## 이것은 `one(eltype (a))` 가 생성 가능하다고 가정하기 때문에 불충분합니다.:
    # R = typeof(op(one(eltype(a)), one(eltype(b))))

    ## 이것은`a [1]`이 있다고 가정하기 때문에 실패하고, 배열의 모든 요소를 나타내기 때문에 실패합니다
    # R = typeof(op(a[1], b[1]))

    ## 이것은 `+` 가 `promote_type` 를 호출한다고 가정하기 때문에 올바르지 않습니다.
    ## 그러나 Bool과 같은 일부 유형에서는 그렇지 않습니다.:
    # R = promote_type(ai, bi)

    # 타입 추론의 반환 값에 따라 매우 취약하기 때문에 잘못되었습니다.(최적화가 불가능할뿐만 아니라):
    # R = Base.return_types(op, (eltype(a), eltype(b)))

    ## 그래서 결국 이렇습니다.:
    R = promote_op(op, eltype(a), eltype(b))
    ## 때로는 원하는 유형보다 더 큰 유형을 제공 할 수도 있지만 항상 올바른 유형을 제공합니다.

    output = similar(b, R, (size(a, 1), size(b, 2)))
    if size(a, 2) > 0
        for j in 1:size(b, 2)
            for i in 1:size(a, 1)
                ## 여기서 `R` 는 `Any`, `zero(Any)` 는 정의되지 않았기 때문에 `ab = zero(R)` 을 사용하지 않습니다.
                ## 우리는 또한 typeof (a * b)! = typeof (a * b + a * b) == R이 가능하기 때문에 `ab::R` 을 선언하여
                ↪  루프에서  `ab` 의 타입을 상수로 만들어야합니다.
                ab::R = a[i, 1] * b[1, j]
                for k in 2:size(a, 2)
                    ab += a[i, k] * b[k, j]
                end
                output[i, j] = ab
            end
        end
```

```
    end
    return output
end
```

별도의 변환과 커널로직

컴파일 시간을 줄이고 복잡성을 테스트하는 한 가지 방법은 원하는 유형으로 변환하기위한 로직과 계산을 분리하는 것입니다. 이를 통해 컴파일러는 변환 논리를 대형 커널의 나머지 본문과 독립적으로 특수화하고 인라인 할 수 있습니다.

이것은 더 큰 클래스 유형에서 변환 할 때 나타나는 공통 패턴입니다 알고리즘이 실제로 지원하는 특정 인수 유형으로 변환합니다.

```
complexfunction(arg::Int) = ...
complexfunction(arg::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

12.6 매개변수적으로 제한된 Varargs 메서드

함수매개변수는 다중인자 함수에 제공 되는 인자의 수를 제한하는 데 사용될 수도 있습니다. `Vararg {T, N}` 표기법은 그러한 제약을 나타내기 위해 사용됩니다. 예:

```
julia> bar(a,b,x::Vararg{Any,2}) = (a,b,x)
bar (generic function with 1 method)

julia> bar(1,2,3)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64)
Closest candidates are:
  bar(::Any, ::Any, ::Any, !Matched::Any) at none:1

julia> bar(1,2,3,4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64, ::Int64, ::Int64)
Closest candidates are:
  bar(::Any, ::Any, ::Any, ::Any) at none:1
```

보다 유용하게 파라미터에 의해 varargs 메서드를 제한하는 것이 가능합니다. 예 :

```
+function getindex(A::AbstractArray{T,N}, indices::Vararg{Number,N}) where {T,N}
```

`indices` 의 수가 배열의 차원과 일치 할 때만 호출됩니다.

오직 제공된 인자의 타입만이 제약을 받아야 할 때 `Vararg{T}` 는 `T...` 와 같이 쓰일 수 있습니다. 예를 들어, `f(x::Int ...) = x` 는 `f(x::Vararg{Int}) = x` 의 속기입니다.

12.7 키워드 인수 선택 사항에 대한 참고 사항

Functions 에서 간략하게 언급했듯이 선택적 인수는 여러 메서드정의의 구문으로 구현됩니다. 예를 들어, 이 정의는 다음과 같습니다.

```
f(a=1,b=2) = a+2b
```

다음 세 가지 방법으로 변환됩니다.

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

This means that calling `f()` is equivalent to calling `f(1,2)`. In this case the result is `5`, because `f(1,2)` invokes the first method of `f` above. However, this need not always be the case. If you define a fourth method that is more specialized for integers: 이것은 `f()` 를 호출하는 것이 `f(1,2)` 를 호출하는 것과 동일하다는 것을 의미합니다.이 경우 결과는5입니다. 왜냐하면 `f(1,2)` 가 위의 `f` 의 첫 번째 메서드를 호출하기 때문입니다. 그러나, 항상 그런 것은 아닙니다. 정수에보다 특수화 된 네 번째 메서드를 정의하면 다음과 같습니다.

```
f(a::Int,b::Int) = a-2b
```

`f()` 와 `f(1,2)` 의 결과는 `-3` 입니다. 즉, 선택적 인수는 해당함수의 특정메서드가 아닌 함수에 연결됩니다. 그것은 메서드가 불려지는 옵션 인수의 형태에 의존합니다. 선택적 인수가 전역 변수로 정의되면 선택적 인수의 유형이 런타임에 변경 될 수도 있습니다.

키워드 인수는 일반적인 위치 인수와는 상당히 다르게 작동합니다. 특히, 메서드 디스패치에 참여하지 않습니다. 메서드는 일치하는 메서드가 식별 된 후에 처리되는 키워드 인수를 가진 위치인수에 기반하여 발송됩니다.

12.8 함수같은 객체

메서드는 형식과 관련이 있으므로 형식에 메서드를 추가하여 임의의 줄리아 개체를 "호출가능" 하게 만들 수 있습니다. (이러한 "호출가능"한 객체를 "functors"라고합니다.)

예를 들어 다항식의 계수를 저장하는 유형을 정의 할 수 있습니다. 다항식을 계산하는 함수 :

```
julia> struct Polynomial{R}
           coeffs::Vector{R}
       end

julia> function (p::Polynomial)(x)
           v = p.coeffs[end]
           for i = (length(p.coeffs)-1):-1:1
               v = v*x + p.coeffs[i]
           end
           return v
       end

julia> (p::Polynomial)() = p(5)
```

함수는 이름 대신 타입으로 지정됩니다. As with normal functions there is a terse syntax form. 함수 본문에서 `p` 는 호출된 객체를 나타냅니다. **다항식** 은 다음과 같이 사용할 수 있습니다:

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> p(3)
931

julia> p()
2551
```

이 메카니즘은 타입 생성자와 클로저 (주변 환경을 참조하는 내부 함수)가 줄리아에서 어떻게 작동 하는지를 결정하는 열쇠이기도합니다.

12.9 빈 일반 함수

때때로 메서드를 추가하지 않고 일반함수를 도입하는 것이 유용하기도 합니다. 이것은 인터페이스 정의와 구현을 분리하는데 사용할 수 있습니다. 문서화 또는 코드가독성을 위해 수행 될 수도 있습니다. 이를위한 문법은 인자의 튜플이없는 빈`function` 블록입니다 :

```
function emptyfunc
end
```

12.10 방법 설계 및 모호한 방지

줄리아의 방법 다형성은 가장 강력한 기능 중 하나이지만,이 힘을 이용하는 것은 설계상의 어려움을 야기 할 수 있습니다. 특히,보다 복잡한 메서드 계층 구조에서는 모호성 (ambiguities) 이 발생하는 경우는 드뭅니다. 위에서, 모호성을 해결할 수 있다고 지적했었습니다.

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

메서드를 정의함으로서

```
f(x::Int, y::Int) = 3
```

이것은 종종 올바른 전략일수 있습니다. 그러나이 조언을 맹목적으로 따르는 것이 비생산적일 수 있는 상황이 있습니다. 특히, 일반 함수의 메서드가 많을수록 모호성 가능성이 커집니다. 메서드 계층구조가 간단한 예제보다 복잡해지면 대체 전략에 대해 신중하게 생각하는 것이 가치가 있습니다. 아래에서는 특정 문제와 이러한 문제를 해결할 수있는 몇 가지 대안을 논의합니다.

튜플 및 NTuple 인수

`Tuple` (그리고 `NTuple`) 인자는 특별한 도전을 제시합니다. 예를 들어,

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

`N==0`일 가능성 때문에 모호하다면 `:Int` 나 `Float64` 변형이 호출되어야 하는지를 결정하는 요소가 없습니다. 모호성을 해결하기 위해 한 가지 방법은 빈 튜플에 대한 메서드를 정의하는 것입니다.

```
f(x::Tuple{}) = 3
```

또는 하나의 메서드를 제외한 모든 메서드에 대해 적어도 하나의 요소가 튜플에 있다고 주장 할 수 있습니다.

```
f(x::NTuple{N,Int}) where {N} = 1           # 이것은 대체품입니다.
f(x::Tuple{Float64, Vararg{Float64}}) = 2   # 적어도 하나의 Float64가 필요합니다.
```

디자인 직교화

이상의 인수를 디스패치하려는 상황일 때 "wrapper"기능이 보다 단순한 설계를 할 수 있는지 고려해야합니다. 예를 들어 여러 변형을 작성하는 대신에 :

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

정의하는 것을 고려할 수 있습니다.

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

여기서 `g` 는 인수를 타입 A로 변환합니다. 이것은 별도의 개념이 별도의 방법으로 할당된 직교 설계(orthogonal design)의 보다 일반적인 매우 구체적인 예입니다. 여기서 `g` 는 대개 fallback 정의를 요구합니다

```
g(x::A) = x
```

이와 관련된 전략은 x와y를 일반적인 유형으로 유도하기 위해 `promote` 을 이용합니다:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

이 디자인의 한 가지 위험은 `x` 와 `y` 를 같은 유형으로 변환하는 적절한 프로모션 방법이 없으면 두 번째 방법이 무한히 반복되어 스택 오버플로가 트리거 될 수 있다는 것입니다.

한 번에 하나의 인수로 디스패치

여러 인수를 전달해야하거나 가능한 많은 변형을 정의하기 위해 너무 많은 조합을 사용하는 많은 대체 시스템이있는 경우에는 "name cascade"를 도입하여 (예를 들어) 첫 번째 인수로 전달한 다음 내부 메서드를 호출하는 것을 고려하십시오:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

그러면 내부 메서드 `_fA` 와 `_fB` 는 `x` 에 대해 서로 모호한 점을 고려하지 않고 `y` 에 디스패치 할 수 있습니다.

이 전략에는 최소 하나이상의 주요한 단점이 있습니다 : 많은 경우, 사용자가 exported한 함수 `f` 의 추가 특수화를 정의하여 `f` 의 동작을 추가로 사용자정의 할 수 없습니다. 대신, 내부 메서드 `_fA` 와 `_fB` 에 대한 전문화를 정의해야하며, 이는 exported한 메서드와 exported한 메서드 사이의 줄을 흐리게 만듭니다.

추상 컨테이너 및 요소 유형

가능한, 파견하는 방법을 정의하지 않도록하십시오. 추상 컨테이너의 특정 요소 유형은 예를 들어,

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

메서드를 정의할 누군가를 위해 모호한 메서드를 정의합니다.

```
-(A::MyArrayType{T}, b::T) where {T}
```

가장 좋은 방법은 다음중 하나의 방법을 정의하는 것을 피하는 것입니다. 대신, 일반적인 메서드`- A::AbstractArray, b)` 이 메서드가 각 컨테이너 유형과 요소 유형에 대해 올바른 작업을 수행하는 일반 호출(예 `:similar` and `-`)로 구현되었는지 확인하십시오. 이것은 당신의 메서드를 직교화 하는 조언의 좀 더 복잡한 변형입니다.

이 접근법이 가능하지 않을 때 모호성을 해결하는 것에 대해 다른 개발자와 토론을 시작하는 것이 좋습니다. 처음에 하나의 방법이 정의되었다고해서 그것이 반드시 수정되거나 제거 될 수 없다는 것을 의미하지는 않습니다. 최후의 수단으로 한 개발자는 "반창고(band-aid)" 방법을 정의 할 수 있습니다

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

그것은 무차별한 힘에 의한 모호성을 해결합니다.

복잡한 인수 "cascades"와 기본 인수

기본값을 제공하는 "cascades"메서드를 정의하는 경우 잠재적 기본값에 해당하는 인수를 삭제하는 데 주의해야합니다. 예를 들어, 디지털 필터링 알고리즘을 작성 중이며 패딩을 적용하여 신호의 가장자리를 처리하는 방법이 있다고 가정합니다:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # 이제 "실제"계산을 수행하십시오.
end
```

이것은 디폴트 패딩을 제공하는 메서드와 충돌 할 것이다:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by default
```

이 두 가지 방법은 A가 계속 커지면서 무한 재귀를 생성합니다.

더 나은 디자인은 다음과 같이 호출 계층 구조를 정의하는 것입니다.

```
struct NoPad end  # 패딩이 필요하지 않거나 이미 적용되었음을 나타냅니다.

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # 기본 경계 조건

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # 새로운 경계 조건을 나타냅니다.
end

# 다른 패딩 방법은 여기에 있습니다.

function myfilter(A, kernel, ::NoPad)
    # 다음은 핵심 계산의 "실제" 구현입니다.
end
```

`NoPad`는 다른 종류의 패딩과 같은 인수 위치에서 제공되기 때문에, 디스패치 계층을 잘 정리하고 애매하게 만들 가능성이 낮습니다. 또한, "public" `myfilter` 인터페이스를 확장합니다. 패딩을 명시 적으로 제어하려는 사용자는 `NoPad` 변형을 직접 호출 할 수 있습니다.

[2] Arthur C. Clarke, Profiles of the Future (1961): Clarke's Third Law.

Chapter 13

Constructors

Constructors [1] are functions that create new objects – specifically, instances of Composite Types. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```
julia> struct Foo
           bar
           baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. However, in some cases more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. Recursive data structures, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

[1] Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse

13.1 Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for `Foo` objects that takes only one argument and uses the given value for both the `bar` and `baz` fields. This is simple:

```
julia> Foo(x) = Foo(x,x)
Foo

julia> Foo(1)
Foo(1, 1)
```

You could also add a zero-argument `Foo` constructor method that supplies default values for both of the `bar` and `baz` fields:

```
julia> Foo() = Foo(0)
Foo

julia> Foo()
Foo(0, 0)
```

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called outer constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

13.2 Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs inner constructor methods. An inner constructor method is like an outer constructor method, except for two differences:

terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from the context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.

2. It has access to a special locally existent function called `new` that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

```
julia> struct OrderedPair
           x::Real
           y::Real
           OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
       end
```

Now `OrderedPair` objects can only be constructed such that `x <= y`:

```
julia> OrderedPair(1, 2)
OrderedPair(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] OrderedPair(::Int64, ::Int64) at ./none:4
 [3] top-level scope
```

If the type were declared `mutable`, you could reach in and directly change the field values to violate this invariant. Of course, messing around with an object's internals uninvited is bad practice. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to `new`, returning the resulting object:

```
julia> struct Foo
           bar
           baz
           Foo(bar,baz) = new(bar,baz)
       end
```

This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

```
julia> struct T1
           x::Int64
       end

julia> struct T2
           x::Int64
           T2(x) = new(x)
       end

julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
T1(1)

julia> T2(1.0)
T2(1)
```

It is good practice to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

13.3 Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
       end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `a` is an instance of `SelfReferential`, then a second instance can be created by the call:

```
julia> b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, is another attempt at defining the `SelfReferential` type, this time using a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
           SelfReferential() = (x = new(); x.obj = x)
       end
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
julia> x = SelfReferential();

julia> x === x
true
```

```
julia> x === x.obj
true

julia> x === x.obj.obj
true
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, it is possible to return incompletely initialized objects:

```
julia> mutable struct Incomplete
           data
           Incomplete() = new()
       end

julia> z = Incomplete();
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

```
julia> z.data
ERROR: UndefRefError: access to undefined reference
```

This avoids the need to continually check for `null` values. However, not all object fields are references. Julia considers some types to be "plain data", meaning all of their data is self-contained and does not reference other objects. The plain data types consist of primitive types (e.g. `Int`) and immutable structs of other plain data types. The initial contents of a plain data type is undefined:

```
julia> struct HasPlain
           n::Int
           HasPlain() = new()
       end

julia> HasPlain()
HasPlain(438103441441)
```

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

```
julia> mutable struct Lazy
           data
           Lazy(v) = complete_me(new(), v)
       end
```

As with incomplete objects returned from constructors, if `complete_me` or any of its callees try to access the `data` field of the `Lazy` object before it has been initialized, an error will be thrown immediately.

13.4 Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from Parametric Types that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

```
julia> struct Point{T<:Real}
           x::T
           y::T
       end

julia> Point(1,2) ## implicit T ##
Point{Int64}(1, 2)

julia> Point(1.0,2.5) ## implicit T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## implicit T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, ::T) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## explicit T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0,2.5) ## explicit T ##
ERROR: InexactError: Int64(2.5)
Stacktrace:
[...]

julia> Point{Float64}(1.0, 2.5) ## explicit T ##
```

```
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1,2) ## explicit T ##
Point{Float64}(1.0, 2.0)
```

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: `Point{Int64}(1,2)` works, but `Point{Int64}(1.0,2.5)` raises an `InexactError` when converting `2.5` to `Int64`. When the type is implied by the arguments to the constructor call, as in `Point(1,2)`, then the types of the arguments must agree – otherwise the `T` cannot be determined – but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What's really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type `T`. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`, that behaves just like non-parametric default inner constructors do. It also provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

```
julia> struct Point{T<:Real}
           x::T
           y::T
           Point{T}(x,y) where {T<:Real} = new(x,y)
       end

julia> Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

Notice that each definition looks like the form of constructor call that it handles. The call `Point{Int64}(1,2)` will invoke the definition `Point{T}(x,y)` inside the `struct` block. The outer constructor declaration, on the other hand, defines a method for the general `Point` constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `Point(1,2)` and `Point(1.0,2.5)`, work. Since the method declaration restricts the arguments to being of the same type, calls like `Point(1,2.5)`, with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `Point(1,2.5)` work by "promoting" the integer value `1` to the floating-point value `1.0`. The simplest way to achieve this is to define the following additional outer constructor method:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

This method uses the `convert` function to explicitly convert `x` to `Float64` and then delegates construction to the general constructor for the case where both arguments are `Float64`. With this method definition what was previously a `MethodError` now successfully creates a point of type `Point{Float64}`:

```
julia> Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(ans)
Point{Float64}
```

However, other similar calls still don't work:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
Closest candidates are:
  Point(::T, !Matched::T) where T<:Real at none:1
```

For a more general way to make all such calls work sensibly, see Conversion and Promotion. At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

The `promote` function converts all its arguments to a common type – in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

13.5 Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, we implement our own rational number type `OurRational`, similar to Julia's built-in `Rational` type, defined in `rational.jl`:

```
julia> struct OurRational{T<:Integer} <: Real
           num::T
           den::T
           function OurRational{T}(num::T, den::T) where T<:Integer
               if num == 0 && den == 0
                    error("invalid rational: 0//0")
               end
               g = gcd(den, num)
               num = div(num, g)
               den = div(den, g)
               new(num, den)
           end
       end

julia> OurRational(n::T, d::T) where {T<:Integer} = OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) = OurRational(promote(n,d)...)
OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational

julia> ⊘(n::Integer, d::Integer) = OurRational(n,d)
⊘ (generic function with 1 method)

julia> ⊘(x::OurRational, y::Integer) = x.num ⊘ (x.den*y)
⊘ (generic function with 2 methods)

julia> ⊘(x::Integer, y::OurRational) = (x*y.den) ⊘ y.num
⊘ (generic function with 3 methods)

julia> ⊘(x::Complex, y::Real) = complex(real(x) ⊘ y, imag(x) ⊘ y)
⊘ (generic function with 4 methods)
```

```
julia> ⊘(x::Real, y::Complex) = (x*y') ⊘ real(y*y')
⊘ (generic function with 5 methods)

julia> function ⊘(x::Complex, y::Complex)
           xy = x*y'
           yy = real(y*y')
           complex(real(xy) ⊘ yy, imag(xy) ⊘ yy)
       end
⊘ (generic function with 6 methods)
```

The first line – `struct OurRational{T<:Integer} <: Real` – declares that `OurRational` takes one type parameter of an integer type, and is itself a real type. The field declarations `num::T` and `den::T` indicate that the data held in a `OurRational{T}` object are a pair of integers of type `T`, one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. `OurRational` has a single inner constructor method which checks that both of `num` and `den` aren't zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by dividing the given numerator and denominator values by their greatest common divisor, computed using the `gcd` function. Since `gcd` returns the greatest common divisor of its arguments with sign matching the first argument (`den` here), after this division the new value of `den` is guaranteed to be non-negative. Because this is the only inner constructor for `OurRational`, we can be certain that `OurRational` objects are always constructed in this normalized form.

`OurRational` also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter `T` from the type of the numerator and denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of `1` as the denominator.

Following the outer constructor definitions, we defined a number of methods for the `⊘` operator, which provides a syntax for writing rationals (e.g. `1 ⊘ 2`). Julia's `Rational` type uses the `//` operator for this purpose. Before these definitions, `⊘` is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in 유리수 – its entire behavior is defined in these few lines. The first and most basic definition just makes `a ⊘ b` construct a `OurRational` by applying the `OurRational` constructor to `a` and `b` when they are integers. When one of the operands of `⊘` is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying `⊘` to complex integral values creates an instance of `Complex{OurRational}` – a complex number whose real and imaginary parts are rationals:

```
julia> z = (1 + 2im) ⊘ (1 - 2im);

julia> typeof(z)
Complex{OurRational{Int64}}

julia> typeof(z) <: Complex{OurRational}
false
```

Thus, although the `⊘` operator usually returns an instance of `OurRational`, if either of its arguments are complex integers, it will return an instance of `Complex{OurRational}` instead. The interested reader should consider perusing the rest of `rational.jl`: it is short, self-contained, and implements an entire basic Julia type.

13.6 Outer-only constructors

As we have seen, a typical parametric type has inner constructors that are called when type parameters are known; e.g. they apply to `Point{Int}` but not to `Point`. Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a `Point{Int}` from the call `Point(1,2)`. Outer constructors call inner constructors to actually make instances. However, in some cases one would rather not provide inner constructors, so that specific type parameters cannot be requested manually.

For example, say we define a type that stores a vector along with an accurate representation of its sum:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32,Int32}(Int32[1, 2, 3], 6)
```

The problem is that we want `S` to be a larger type than `T`, so that we can sum many elements with less information loss. For example, when `T` is `Int32`, we would like `S` to be `Int64`. Therefore we want to avoid an interface that allows the user to construct instances of the type `SummedArray{Int32,Int32}`. One way to do this is to provide a constructor only for `SummedArray`, but inside the `struct` definition block to suppress generation of default constructors:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
           function SummedArray(a::Vector{T}) where T
```

```
            S = widen(T)
            new{T,S}(a, sum(S, a))
        end
    end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
ERROR: MethodError: no method matching SummedArray(::Array{Int32,1}, ::Int32)
Closest candidates are:
  SummedArray(::Array{T,1}) where T at none:5
```

This constructor will be invoked by the syntax `SummedArray(a)`. The syntax `new{T,S}` allows specifying parameters for the type to be constructed, i.e. this call will return a `SummedArray{T,S}`. `new{T,S}` can be used in any constructor definition, but for convenience the parameters to `new{}` are automatically derived from the type being constructed when possible.

Chapter 14

Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including 정수와 부동 소수점 수, 산술 연산과 기본 함수, Types, and Methods. In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

- Automatic promotion for built-in arithmetic types and operators. In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as `+`, `-`, `*`, and `/`, are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum `1 + 1.5` as the floating-point value `2.5`, even though one of the operands to `+` is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion before addition since integers and floating-point values cannot be added as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.
- No automatic promotion. This camp includes Ada and ML – very "strict" statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression `1 + 1.5` would be a compilation error in both Ada and ML. Instead one must write `real(1) + 1.5`, explicitly converting the integer `1` to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the "no automatic promotion" category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that

applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch – something which Julia's dispatch and type systems are particularly well-suited to handle. "Automatic" promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they should promote to when mixed with other types.

14.1 Conversion

The standard way to obtain a value of a certain type `T` is to call the type's constructor, `T(x)`. However, there are cases where it's convenient to convert a value from one type to another without the programmer asking for it explicitly. One example is assigning a value into an array: if `A` is a `Vector{Float64}`, the expression `A[1] = 2` should work by automatically converting the `2` from `Int` to `Float64`, and storing the result in the array. This is done via the `convert` function.

The `convert` function generally takes two arguments: the first is a type object and the second is a value to convert to that type. The returned value is the value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(UInt8, x)
0x0c

julia> typeof(ans)
UInt8

julia> convert(AbstractFloat, x)
12.0

julia> typeof(ans)
Float64
```

```
julia> a = Any[1 2 3; 4 5 6]
2×3 Array{Any,2}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

Conversion isn't always possible, in which case a no method error is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type AbstractFloat
[...]
```

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically), however Julia does not: even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are. Therefore in Julia the dedicated `parse` function must be used to perform this operation, making it more explicit.

When is `convert` called?

The following language constructs call `convert`:

- Assigning to an array converts to the array's element type.
- Assigning to a field of an object converts to the declared type of the field.
- Constructing an object with `new` converts to the object's declared field types.
- Assigning to a variable with a declared type (e.g. `local x::T`) converts to that type.
- A function with a declared return type converts its return value to that type.
- Passing a value to `ccall` converts it to the corresponding argument type.

Conversion vs. Construction

Note that the behavior of `convert(T, x)` appears to be nearly identical to `T(x)`. Indeed, it usually is. However, there is a key semantic difference: since `convert` can be called implicitly, its methods are restricted to cases that are considered "safe" or "unsurprising". `convert` will only convert between types that represent the same basic kind of thing (e.g. different representations of numbers, or different string encodings). It is also usually lossless; converting a value to a different type and back again should result in the exact same value.

There are four general kinds of cases where constructors differ from `convert`:

Constructors for types unrelated to their arguments

Some constructors don't implement the concept of "conversion". For example, `Timer(2)` creates a 2-second timer, which is not really a "conversion" from an integer to a timer.

Mutable collections

`convert(T, x)` is expected to return the original `x` if `x` is already of type `T`. In contrast, if `T` is a mutable collection type then `T(x)` should always make a new collection (copying elements from `x`).

Wrapper types

For some types which "wrap" other values, the constructor may wrap its argument inside a new object even if it is already of the requested type. For example `Some(x)` wraps `x` to indicate that a value is present (in a context where the result might be a `Some` or `nothing`). However, `x` itself might be the object `Some(y)`, in which case the result is `Some(Some(y))`, with two levels of wrapping. `convert(Some, x)`, on the other hand, would just return `x` since it is already a `Some`.

Constructors that don't return instances of their own type

In very rare cases it might make sense for the constructor `T(x)` to return an object not of type `T`. This could happen if a wrapper type is its own inverse (e.g. `Flip(Flip(x)) === x`), or to support an old calling syntax for backwards compatibility when a library is restructured. But `convert(T, x)` should always return a value of type `T`.

Defining New Conversions

When defining a new type, initially all ways of creating it should be defined as constructors. If it becomes clear that implicit conversion would be useful, and that some constructors meet the above "safety" criteria, then `convert` methods can be added. These methods are typically quite simple, as they only need to call the appropriate constructor. Such a definition might look like this:

```
convert(::Type{MyType}, x) = MyType(x)
```

The type of the first argument of this method is a singleton type, `Type{MyType}`, the only instance of which is `MyType`. Thus, this method is only invoked when the first argument is the type value `MyType`. Notice the syntax used for the first argument: the argument name is omitted prior to the `::` symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value does not need to be referenced by name. In this example, since the type is a singleton, we already know its value without referring to an argument name.

All instances of some abstract types are by default considered "sufficiently similar" that a universal `convert` definition is provided in Julia Base. For example, this definition states that it's valid to `convert` any `Number` type to any other by calling a 1-argument constructor:

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)
```

This means that new `Number` types only need to define constructors, since this definition will handle `convert` for them. An identity conversion is also provided to handle the case where the argument is already of the requested type:

```
convert(::Type{T}, x::T) where {T<:Number} = x
```

Similar definitions exist for `AbstractString`, `AbstractArray`, and `AbstractDict`.

14.2 Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term "promotion" is appropriate since the values are converted to a "greater" type — i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every `Int32` value can also be represented as a `Float64` value, `Int32` is not a subtype of `Float64`.

Promotion to a common "greater" type is performed in Julia by the `promote` function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
```

```
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the larger of either the native machine word size or the largest integer argument type. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical "clever" application being the definition of catch-all methods for numeric operations like the arithmetic operators `+`, `-`, `*` and `/`. Here are some of the catch-all method definitions given in `promotion.jl`:

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations – it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in `promotion.jl`, but beyond that, there are hardly any calls to `promote` required in Julia Base. The most common usages of `promote` occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that `rational.jl` provides the following outer constructor method:

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(ans)
Rational{Int32}
```

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

Defining Promotion Rules

Although one could, in principle, define methods for the `promote` function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of `promote` is defined in terms of an auxiliary function called `promote_rule`, which one can provide methods for. The `promote_rule` function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

```
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types, however; the following promotion rules both occur in Julia Base:

```
promote_rule(::Type{BigInt}, ::Type{Float64}) = BigFloat
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

In the latter case, the result type is `BigInt` since `BigInt` is the only type large enough to hold integers for arbitrary-precision integer arithmetic. Also note that one does not need to define both `promote_rule(::Type{A}, ::Type{B})` and `promote_rule(::Type{B}, ::Type{A})` – the symmetry is implied by the way `promote_rule` is used in the promotion process.

The `promote_rule` function is used as a building block to define a second function called `promote_type`, which, given any number of type objects, returns the common type to which those values, as arguments to `promote` should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use `promote_type`:

```
julia> promote_type(Int8, Int64)
Int64
```

Internally, `promote_type` is used inside of `promote` to determine what type argument values should be converted to for promotion. It can, however, be useful in its own right. The curious reader can read the code in `promotion.jl`, which defines the complete promotion mechanism in about 35 lines.

Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

```
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} = Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where {T<:Integer,S<:Integer} = Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:AbstractFloat} = promote_type(T,S)
```

The first rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of promotion of its numerator/denominator type with the other integer type. The second rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The third and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the type's constructors and the default `convert` method for numbers, are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types – integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

Chapter 15

Interfaces

A lot of the power and extensibility in Julia comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors.

15.1 Iteration

Required methods		Brief description
`iterate(iter)`		Returns either a tuple of the first item and initial state or `nothing` if empty
`iterate(iter, state)`		Returns either a tuple of the next item and next state or `nothing` if no items remain
Important optional methods	Default definition	Brief description
`IteratorSize(IterType)`	`HasLength()`	One of `HasLength()`, `HasShape{N}()`, `IsInfinite()`, or `SizeUnknown()` as appropriate
`IteratorEltype(IterType)`	`HasEltype()`	Either `EltypeUnknown()` or `HasEltype()` as appropriate
`eltype(IterType)`	`Any`	The type of the first entry of the tuple returned by `iterate()`
`length(iter)`	(undefined)	The number of items, if known
`size(iter, [dim])`	(undefined)	The number of items in each dimension, if known

Sequential iteration is implemented by the `iterate` function. Instead of mutating objects as they are iterated over, Julia iterators may keep track of the iteration state externally from the object. The return value from iterate is always either a tuple of a value and a state, or `nothing` if no elements remain. The state object will be passed back to the iterate function on the next iteration and is generally considered an implementation detail private to the iterable object.

Value returned by `IteratorSize(IterType)`	Required Methods
`HasLength()`	`length(iter)`
`HasShape{N}()`	`length(iter)` and `size(iter, [dim])`
`IsInfinite()`	(none)
`SizeUnknown()`	(none)

Value returned by `IteratorEltype(IterType)`	Required Methods
`HasEltype()`	`eltype(IterType)`
`EltypeUnknown()`	(none)

Any object that defines this function is iterable and can be used in the many functions that rely upon iteration. It can also be used directly in a `for` loop since the syntax:

```
for i in iter   # or  "for i = iter"
    # body
end
```

is translated into:

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

A simple example is an iterable sequence of square numbers with a defined length:

```
julia> struct Squares
           count::Int
       end

julia> Base.iterate(S::Squares, state=1) = state > S.count ? nothing : (state*state, state+1)
```

With only `iterate` definition, the `Squares` type is already pretty powerful. We can iterate over all the elements:

```
julia> for i in Squares(7)
           println(i)
       end
1
4
9
16
25
36
49
```

We can use many of the builtin methods that work with iterables, like `in`, or `mean` and `std` from the `Statistics` standard library module:

```
julia> 25 in Squares(10)
true

julia> using Statistics

julia> mean(Squares(100))
3383.5

julia> std(Squares(100))
3024.355854282583
```

There are a few more methods we can extend to give Julia more information about this iterable collection. We know that the elements in a `Squares` sequence will always be `Int`. By extending the `eltype` method, we can give that information to Julia and help it make more specialized code in the more complicated methods. We also know the number of elements in our sequence, so we can extend `length`, too:

```
julia> Base.eltype(::Type{Squares}) = Int # Note that this is defined for the type

julia> Base.length(S::Squares) = S.count
```

Now, when we ask Julia to `collect` all the elements into an array it can preallocate a `Vector{Int}` of the right size instead of blindly `push!`ing each element into a `Vector{Any}`:

```
julia> collect(Squares(4))
4-element Array{Int64,1}:
```

```
 1
 4
 9
16
```

While we can rely upon generic implementations, we can also extend specific methods where we know there is a simpler algorithm. For example, there's a formula to compute the sum of squares, so we can override the generic iterative version with a more performant solution:

```
julia> Base.sum(S::Squares) = (n = S.count; return n*(n+1)*(2n+1)÷6)

julia> sum(Squares(1803))
1955361914
```

This is a very common pattern throughout Julia Base: a small set of required methods define an informal interface that enable many fancier behaviors. In some cases, types will want to additionally specialize those extra behaviors when they know a more efficient algorithm can be used in their specific case.

It is also often useful to allow iteration over a collection in reverse order by iterating over `Iterators.reverse(iterator)`. To actually support reverse-order iteration, however, an iterator type `T` needs to implement `iterate` for `Iterators.Reverse{T}`. (Given `r::Iterators.Reverse{T}`, the underling iterator of type `T` is `r.itr`.) In our `Squares` example, we would implement `Iterators.Reverse{Squares}` methods:

```
julia> Base.iterate(rS::Iterators.Reverse{Squares}, state=rS.itr.count) = state < 1 ? nothing : (state*state,
↪  state-1)

julia> collect(Iterators.reverse(Squares(4)))
4-element Array{Int64,1}:
 16
  9
  4
  1
```

15.2 Indexing

For the `Squares` iterable above, we can easily compute the `i`th element of the sequence by squaring it. We can expose this as an indexing expression `S[i]`. To opt into this behavior, `Squares` simply needs to define `getindex`:

Methods to implement	Brief description
`getindex(X, i)`	`X[i]`, indexed element access
`setindex!(X, v, i)`	`X[i] = v`, indexed assignment
`firstindex(X)`	The first index
`lastindex(X)`	The last index, used in `X[end]`

```
julia> function Base.getindex(S::Squares, i::Int)
           1 <= i <= S.count ¦¦ throw(BoundsError(S, i))
           return i*i
       end

julia> Squares(100)[23]
529
```

Additionally, to support the syntax `S[end]`, we must define `lastindex` to specify the last valid index. It is recommended to also define `firstindex` to specify the first valid index:

```
julia> Base.firstindex(S::Squares) = 1

julia> Base.lastindex(S::Squares) = length(S)

julia> Squares(23)[end]
529
```

Note, though, that the above only defines `getindex` with one integer index. Indexing with anything other than an `Int` will throw a `MethodError` saying that there was no matching method. In order to support indexing with ranges or vectors of `Int`s, separate methods must be written:

```
julia> Base.getindex(S::Squares, i::Number) = S[convert(Int, i)]

julia> Base.getindex(S::Squares, I) = [S[i] for i in I]

julia> Squares(10)[[3,4.,5]]
3-element Array{Int64,1}:
  9
 16
 25
```

While this is starting to support more of the indexing operations supported by some of the builtin types, there's still quite a number of behaviors missing. This `Squares` sequence is starting to look more and more like a vector as we've added behaviors to it. Instead of defining all these behaviors ourselves, we can officially define it as a subtype of an `AbstractArray`.

15.3 Abstract Arrays

If a type is defined as a subtype of `AbstractArray`, it inherits a very large set of rich behaviors including iteration and multidimensional indexing built on top of single-element access. See the arrays manual page and the Julia Base section for more supported methods.

A key part in defining an `AbstractArray` subtype is `IndexStyle`. Since indexing is such an important part of an array and often occurs in hot loops, it's important to make both indexing and indexed assignment as efficient as possible. Array data structures are typically defined in one of two ways: either it most efficiently accesses its elements using just one index (linear indexing) or it intrinsically accesses the elements with indices specified for every dimension. These two modalities are identified by Julia as `IndexLinear()` and `IndexCartesian()`. Converting a linear index to multiple indexing subscripts is typically very expensive, so this provides a traits-based mechanism to enable efficient generic code for all array types.

This distinction determines which scalar indexing methods the type must define. `IndexLinear()` arrays are simple: just define `getindex(A::ArrayType, i::Int)`. When the array is subsequently indexed with a multidimensional set of indices, the fallback `getindex(A::AbstractArray, I...)()` efficiently converts the indices into one linear index and then calls the above method. `IndexCartesian()` arrays, on the other hand, require methods to be defined for each supported dimensionality with `ndims(A)` `Int` indices. For example, `SparseMatrixCSC` from the `SparseArrays` standard library module, only supports two dimensions, so it just defines `getindex(A::SparseMatrixCSC, i::Int, j::Int)`. The same holds for `setindex!`.

Returning to the sequence of squares from above, we could instead define it as a subtype of an `AbstractArray{Int, 1}`:

```
julia> struct SquaresVector <: AbstractArray{Int, 1}
           count::Int
       end

julia> Base.size(S::SquaresVector) = (S.count,)

julia> Base.IndexStyle(::Type{<:SquaresVector}) = IndexLinear()

julia> Base.getindex(S::SquaresVector, i::Int) = i*i
```

Note that it's very important to specify the two parameters of the `AbstractArray`; the first defines the `eltype`, and the second defines the `ndims`. That supertype and those three methods are all it takes for `SquaresVector` to be an iterable, indexable, and completely functional array:

```
julia> s = SquaresVector(4)
4-element SquaresVector:
  1
  4
  9
 16

julia> s[s .> 8]
2-element Array{Int64,1}:
  9
 16

julia> s + s
4-element Array{Int64,1}:
  2
  8
 18
 32

julia> sin.(s)
4-element Array{Float64,1}:
  0.8414709848078965
 -0.7568024953079282
  0.4121184852417566
 -0.2879033166650653
```

As a more complicated example, let's define our own toy N-dimensional sparse-like array type built on top of `Dict`:

```
julia> struct SparseArray{T,N} <: AbstractArray{T,N}
           data::Dict{NTuple{N,Int}, T}
           dims::NTuple{N,Int}
       end

julia> SparseArray(::Type{T}, dims::Int...) where {T} = SparseArray(T, dims);
```

```
julia> SparseArray(::Type{T}, dims::NTuple{N,Int}) where {T,N} = SparseArray{T,N}(Dict{NTuple{N,Int}, T}(), dims);

julia> Base.size(A::SparseArray) = A.dims

julia> Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where {T} = SparseArray(T, dims)

julia> Base.getindex(A::SparseArray{T,N}, I::Vararg{Int,N}) where {T,N} = get(A.data, I, zero(T))

julia> Base.setindex!(A::SparseArray{T,N}, v, I::Vararg{Int,N}) where {T,N} = (A.data[I] = v)
```

Notice that this is an `IndexCartesian` array, so we must manually define `getindex` and `setindex!` at the dimensionality of the array. Unlike the `SquaresVector`, we are able to define `setindex!`, and so we can mutate the array:

```
julia> A = SparseArray(Float64, 3, 3)
3×3 SparseArray{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2)
3×3 SparseArray{Float64,2}:
 2.0  2.0  2.0
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> A[:] = 1:length(A); A
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

The result of indexing an `AbstractArray` can itself be an array (for instance when indexing by an `AbstractRange`). The `AbstractArray` fallback methods use `similar` to allocate an `Array` of the appropriate size and element type, which is filled in using the basic indexing method described above. However, when implementing an array wrapper you often want the result to be wrapped as well:

```
julia> A[1:2,:]
2×3 SparseArray{Float64,2}:
```

```
1.0  4.0  7.0
2.0  5.0  8.0
```

In this example it is accomplished by defining `Base.similar{T}(A::SparseArray, ::Type{T}, dims::Dims)` to create the appropriate wrapped array. (Note that while `similar` supports 1- and 2-argument forms, in most case you only need to specialize the 3-argument form.) For this to work it's important that `SparseArray` is mutable (supports `setindex!`). Defining `similar`, `getindex` and `setindex!` for `SparseArray` also makes it possible to `copy` the array:

```
julia> copy(A)
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

In addition to all the iterable and indexable methods from above, these types can also interact with each other and use most of the methods defined in Julia Base for `AbstractArrays`:

```
julia> A[SquaresVector(3)]
3-element SparseArray{Float64,1}:
 1.0
 4.0
 9.0

julia> sum(A)
45.0
```

If you are defining an array type that allows non-traditional indexing (indices that start at something other than 1), you should specialize `axes`. You should also specialize `similar` so that the `dims` argument (ordinarily a `Dims` size-tuple) can accept `AbstractUnitRange` objects, perhaps range-types `Ind` of your own design. For more information, see Arrays with custom indices.

15.4 Strided Arrays

A strided array is a subtype of `AbstractArray` whose entries are stored in memory with fixed strides. Provided the element type of the array is compatible with BLAS, a strided array can utilize BLAS and LAPACK routines for more efficient linear algebra routines. A typical example of a user-defined strided array is one that wraps a standard `Array` with additional structure.

Warning: do not implement these methods if the underlying storage is not actually strided, as it may lead to incorrect results or segmentation faults.

Here are some examples to demonstrate which type of arrays are strided and which are not:

```
1:5   # not strided (there is no storage associated with this array.)
Vector(1:5)  # is strided with strides (1,)
A = [1 5; 2 6; 3 7; 4 8]  # is strided with strides (1,4)
V = view(A, 1:2, :)   # is strided with strides (1,4)
V = view(A, 1:2:3, 1:2)   # is strided with strides (2,4)
V = view(A, [1,2,4], :)   # is not strided, as the spacing between rows is not fixed.
```

15.5 Customizing broadcasting

Broadcasting is triggered by an explicit call to `broadcast` or `broadcast!`, or implicitly by "dot" operations like `A .+ b` or `f.(x, y)`. Any object that has axes and supports indexing can participate as an argument in broadcasting, and by default the result is stored in an `Array`. This basic framework is extensible in three major ways:

- Ensuring that all arguments support broadcast
- Selecting an appropriate output array for the given set of arguments
- Selecting an efficient implementation for the given set of arguments

Not all types support `axes` and indexing, but many are convenient to allow in broadcast. The `Base.broadcastable` function is called on each argument to broadcast, allowing it to return something different that supports `axes` and indexing. By default, this is the identity function for all `AbstractArray`s and `Number`s — they already support `axes` and indexing. For a handful of other types (including but not limited to types themselves, functions, special singletons like `missing` and `nothing`, and dates), `Base.broadcastable` returns the argument wrapped in a `Ref` to act as a 0-dimensional "scalar" for the purposes of broadcasting. Custom types can similarly specialize `Base.broadcastable` to define their shape, but they should follow the convention that `collect(Base.broadcastable(x)) == collect(x)`. A notable exception is `AbstractString`; strings are special-cased to behave as scalars for the purposes of broadcast even though they are iterable collections of their characters (see Strings for more).

The next two steps (selecting the output array and implementation) are dependent upon determining a single answer for a given set of arguments. Broadcast must take all the varied types of its arguments and collapse them down to just one output array and one implementation. Broadcast calls this single answer a "style." Every broadcastable object each has its own preferred style, and a promotion-like system is used to combine these styles into a single answer — the "destination style".

Broadcast Styles

`Base.BroadcastStyle` is the abstract type from which all broadcast styles are derived. When used as a function it has two possible forms, unary (single-argument) and binary. The unary variant states that you intend to implement specific broadcasting behavior and/or output type, and do not wish to rely on the default fallback `Broadcast.DefaultArrayStyle`.

To override these defaults, you can define a custom `BroadcastStyle` for your object:

```
struct MyStyle <: Broadcast.BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyType}) = MyStyle()
```

In some cases it might be convenient not to have to define `MyStyle`, in which case you can leverage one of the general broadcast wrappers:

- `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.Style{MyType}()` can be used for arbitrary types.
- `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.ArrayStyle{MyType}()` is preferred if `MyType` is an `AbstractArray`.
- For `AbstractArrays` that only support a certain dimensionality, create a subtype of `Broadcast.AbstractArrayStyle{N}` (see below).

When your broadcast operation involves several arguments, individual argument styles get combined to determine a single `DestStyle` that controls the type of the output container. For more details, see below.

Selecting an appropriate output array

The broadcast style is computed for every broadcasting operation to allow for dispatch and specialization. The actual allocation of the result array is handled by `similar`, using the Broadcasted object as its first argument.

```
Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})
```

The fallback definition is

```
similar(bc::Broadcasted{DefaultArrayStyle{N}}, ::Type{ElType}) where {N,ElType} =
    similar(Array{ElType}, axes(bc))
```

However, if needed you can specialize on any or all of these arguments. The final argument `bc` is a lazy representation of a (potentially fused) broadcast operation, a `Broadcasted` object. For these purposes, the most important fields of the wrapper are `f` and `args`, describing the function and argument list, respectively. Note that the argument list can — and often does — include other nested `Broadcasted` wrappers.

For a complete example, let's say you have created a type, `ArrayAndChar`, that stores an array and a single character:

```
struct ArrayAndChar{T,N} <: AbstractArray{T,N}
    data::Array{T,N}
    char::Char
end
Base.size(A::ArrayAndChar) = size(A.data)
Base.getindex(A::ArrayAndChar{T,N}, inds::Vararg{Int,N}) where {T,N} = A.data[inds...]
Base.setindex!(A::ArrayAndChar{T,N}, val, inds::Vararg{Int,N}) where {T,N} = A.data[inds...] = val
Base.showarg(io::IO, A::ArrayAndChar, toplevel) = print(io, typeof(A), " with char '", A.char, "'")
```

You might want broadcasting to preserve the `char` "metadata." First we define

```
Base.BroadcastStyle(::Type{<:ArrayAndChar}) = Broadcast.ArrayStyle{ArrayAndChar}()
```

This means we must also define a corresponding `similar` method:

```
function Base.similar(bc::Broadcast.Broadcasted{Broadcast.ArrayStyle{ArrayAndChar}}, ::Type{ElType}) where ElType
    # Scan the inputs for the ArrayAndChar:
    A = find_aac(bc)
    # Use the char field of A to create the output
    ArrayAndChar(similar(Array{ElType}, axes(bc)), A.char)
end

"`A = find_aac(As)` returns the first ArrayAndChar among the arguments."
find_aac(bc::Base.Broadcast.Broadcasted) = find_aac(bc.args)
find_aac(args::Tuple) = find_aac(find_aac(args[1]), Base.tail(args))
find_aac(x) = x
find_aac(a::ArrayAndChar, rest) = a
find_aac(::Any, rest) = find_aac(rest)
```

From these definitions, one obtains the following behavior:

```
julia> a = ArrayAndChar([1 2; 3 4], 'x')
2×2 ArrayAndChar{Int64,2} with char 'x':
 1  2
 3  4

julia> a .+ 1
2×2 ArrayAndChar{Int64,2} with char 'x':
 2  3
```

```
 4  5


julia> a .+ [5,10]
2×2 ArrayAndChar{Int64,2} with char 'x':
  6   7
 13  14
```

Extending broadcast with custom implementations

In general, a broadcast operation is represented by a lazy `Broadcasted` container that holds onto the function to be applied alongside its arguments. Those arguments may themselves be more nested `Broadcasted` containers, forming a large expression tree to be evaluated. A nested tree of `Broadcasted` containers is directly constructed by the implicit dot syntax; `5 .+ 2.*x` is transiently represented by `Broadcasted(+, 5, Broadcasted(*, 2, x))`, for example. This is invisible to users as it is immediately realized through a call to `copy`, but it is this container that provides the basis for broadcast's extensibility for authors of custom types. The built-in broadcast machinery will then determine the result type and size based upon the arguments, allocate it, and then finally copy the realization of the `Broadcasted` object into it with a default `copyto!(::AbstractArray, ::Broadcasted)` method. The built-in fallback `broadcast` and `broadcast!` methods similarly construct a transient `Broadcasted` representation of the operation so they can follow the same codepath. This allows custom array implementations to provide their own `copyto!` specialization to customize and optimize broadcasting. This is again determined by the computed broadcast style. This is such an important part of the operation that it is stored as the first type parameter of the `Broadcasted` type, allowing for dispatch and specialization.

For some types, the machinery to "fuse" operations across nested levels of broadcasting is not available or could be done more efficiently incrementally. In such cases, you may need or want to evaluate `x .* (x .+ 1)` as if it had been written `broadcast(*, x, broadcast(+, x, 1))`, where the inner operation is evaluated before tackling the outer operation. This sort of eager operation is directly supported by a bit of indirection; instead of directly constructing `Broadcasted` objects, Julia lowers the fused expression `x .* (x .+ 1)` to `Broadcast.broadcasted(*, x, Broadcast.broadcasted(+, x, 1))`. Now, by default, `broadcasted` just calls the `Broadcasted` constructor to create the lazy representation of the fused expression tree, but you can choose to override it for a particular combination of function and arguments.

As an example, the builtin `AbstractRange` objects use this machinery to optimize pieces of broadcasted expressions that can be eagerly evaluated purely in terms of the start, step, and length (or stop) instead of computing every single element. Just like all the other machinery, `broadcasted` also computes and exposes the combined broadcast style of its arguments, so instead of specializing on `broadcasted(f, args...)`, you can specialize on `broadcasted(::DestStyle, f, args...)` for any combination of style, function, and arguments.

For example, the following definition supports the negation of ranges:

```
broadcasted(::DefaultArrayStyle{1}, ::typeof(-), r::OrdinalRange) = range(-first(r), step=-step(r),
↪  length=length(r))
```

Extending in-place broadcasting

In-place broadcasting can be supported by defining the appropriate `copyto!(dest, bc::Broadcasted)` method. Because you might want to specialize either on `dest` or the specific subtype of `bc`, to avoid ambiguities between packages we recommend the following convention.

If you wish to specialize on a particular style `DestStyle`, define a method for

```
copyto!(dest, bc::Broadcasted{DestStyle})
```

Optionally, with this form you can also specialize on the type of `dest`.

If instead you want to specialize on the destination type `DestType` without specializing on `DestStyle`, then you should define a method with the following signature:

```
copyto!(dest::DestType, bc::Broadcasted{Nothing})
```

This leverages a fallback implementation of `copyto!` that converts the wrapper into a `Broadcasted{Nothing}`. Consequently, specializing on `DestType` has lower precedence than methods that specialize on `DestStyle`.

Similarly, you can completely override out-of-place broadcasting with a `copy(::Broadcasted)` method.

Working with `Broadcasted` objects

In order to implement such a `copy` or `copyto!`, method, of course, you must work with the `Broadcasted` wrapper to compute each element. There are two main ways of doing so:

- `Broadcast.flatten` recomputes the potentially nested operation into a single function and flat list of arguments. You are responsible for implementing the broadcasting shape rules yourself, but this may be helpful in limited situations.
- Iterating over the `CartesianIndices` of the `axes(::Broadcasted)` and using indexing with the resulting `CartesianIndex` object to compute the result.

Writing binary broadcasting rules

The precedence rules are defined by binary `BroadcastStyle` calls:

```
Base.BroadcastStyle(::Style1, ::Style2) = Style12()
```

where `Style12` is the `BroadcastStyle` you want to choose for outputs involving arguments of `Style1` and `Style2`. For example,

```
Base.BroadcastStyle(::Broadcast.Style{Tuple}, ::Broadcast.AbstractArrayStyle{0}) = Broadcast.Style{Tuple}()
```

indicates that `Tuple` "wins" over zero-dimensional arrays (the output container will be a tuple). It is worth noting that you do not need to (and should not) define both argument orders of this call; defining one is sufficient no matter what order the user supplies the arguments in.

For `AbstractArray` types, defining a `BroadcastStyle` supersedes the fallback choice, `Broadcast.DefaultArrayStyle`. `DefaultArrayStyle` and the abstract supertype, `AbstractArrayStyle`, store the dimensionality as a type parameter to support specialized array types that have fixed dimensionality requirements.

`DefaultArrayStyle` "loses" to any other `AbstractArrayStyle` that has been defined because of the following methods:

```
BroadcastStyle(a::AbstractArrayStyle{Any}, ::DefaultArrayStyle) = a
BroadcastStyle(a::AbstractArrayStyle{N}, ::DefaultArrayStyle{N}) where N = a
BroadcastStyle(a::AbstractArrayStyle{M}, ::DefaultArrayStyle{N}) where {M,N} =
    typeof(a)(_max(Val(M),Val(N)))
```

You do not need to write binary `BroadcastStyle` rules unless you want to establish precedence for two or more non-`DefaultArrayStyle` types.

If your array type does have fixed dimensionality requirements, then you should subtype `AbstractArrayStyle`. For example, the sparse array code has the following definitions:

```
struct SparseVecStyle <: Broadcast.AbstractArrayStyle{1} end
struct SparseMatStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseVector}) = SparseVecStyle()
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatStyle()
```

Whenever you subtype `AbstractArrayStyle`, you also need to define rules for combining dimensionalities, by creating a constructor for your style that takes a `Val(N)` argument. For example:

```
SparseVecStyle(::Val{0}) = SparseVecStyle()
SparseVecStyle(::Val{1}) = SparseVecStyle()
SparseVecStyle(::Val{2}) = SparseMatStyle()
SparseVecStyle(::Val{N}) where N = Broadcast.DefaultArrayStyle{N}()
```

These rules indicate that the combination of a `SparseVecStyle` with 0- or 1-dimensional arrays yields another `SparseVecStyle`, that its combination with a 2-dimensional array yields a `SparseMatStyle`, and anything of higher dimensionality falls back to the dense arbitrary-dimensional framework. These rules allow broadcasting to keep the sparse representation for operations that result in one or two dimensional outputs, but produce an `Array` for any other dimensionality.

Methods to implement		Brief description
`size(A)`		Returns a tuple containing the dimensions of `A`
`getindex(A, i::Int)`		(if `IndexLinear`) Linear scalar indexing
`getindex(A, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexing
`setindex!(A, v, i::Int)`		(if `IndexLinear`) Scalar indexed assignment
`setindex!(A, v, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexed assignment
Optional methods	Default definition	Brief description
`IndexStyle(::Type)`	`IndexCartesian()`	Returns either `IndexLinear()` or `IndexCartesian()`. See the description below.
`getindex(A, I...)`	defined in terms of scalar `getindex`	Multidimensional and nonscalar indexing
`setindex!(A, I...)`	defined in terms of scalar `setindex!`	Multidimensional and nonscalar indexed assignment
`iterate`	defined in terms of scalar `getindex`	Iteration
`length(A)`	`prod(size(A))`	Number of elements
`similar(A)`	`similar(A, eltype(A), size(A))`	Return a mutable array with the same shape and element type
`similar(A, ::Type{S})`	`similar(A, S, size(A))`	Return a mutable array with the same shape and the specified element type
`similar(A, dims::Dims)`	`similar(A, eltype(A), dims)`	Return a mutable array with the same element type and size dims
`similar(A, ::Type{S}, dims::Dims)`	`Array{S}(undef, dims)`	Return a mutable array with the specified element type and size
Non-traditional indices	Default definition	Brief description
`axes(A)`	`map(OneTo, size(A))`	Return the `AbstractUnitRange` of valid indices
`similar(A, ::Type{S}, inds)`	`similar(A, S, Base.to_shape(inds))`	Return a mutable array with the specified indices `inds` (see below)
`similar(T::Union{Type,Function}, inds)`	`T(Base.to_shape(inds))`	Return an array similar to `T` with the specified indices `inds` (see below)

Methods to implement		Brief description
`strides(A)`		Return the distance in memory (in number of elements) between adjacent elements in each dimension as a tuple. If `A` is an `AbstractArray{T,0}`, this should return an empty tuple.
`Base.unsafe_convert(::Type{Ptr{T}}, A)`		Return the native address of an array.
Optional methods	Default definition	Brief description
`stride(A, i::Int)`	`strides(A)[i]`	Return the distance in memory (in number of elements) between adjacent elements in dimension `k`.

Methods to implement	Brief description
`Base.BroadcastStyle(::Type{SrcType}) = SrcStyle()`	Broadcasting behavior of `SrcType`
`Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})`	Allocation of output container
Optional methods	
`Base.BroadcastStyle(::Style1, ::Style2) = Style12()`	Precedence rules for mixing styles
`Base.axes(x)`	Declaration of the indices of `x`, as per `axes(x)`.
`Base.broadcastable(x)`	Convert `x` to an object that has `axes` and supports indexing
Bypassing default machinery	
`Base.copy(bc::Broadcasted{DestStyle})`	Custom implementation of `broadcast`
`Base.copyto!(dest, bc::Broadcasted{DestStyle})`	Custom implementation of `broadcast!`, specializing on `DestStyle`
`Base.copyto!(dest::DestType, bc::Broadcasted{Nothing})`	Custom implementation of `broadcast!`, specializing on `DestType`
`Base.Broadcast.broadcasted(f, args...)`	Override the default lazy behavior within a fused expression
`Base.Broadcast.instantiate(bc::Broadcasted{DestStyle})`	Override the computation of the lazy broadcast's axes

Chapter 16

Modules

Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope. They are delimited syntactically, inside `module Name ... end`. Modules allow you to create top-level definitions (aka global variables) without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

The following example demonstrates the major features of modules. It is not meant to be run, but is shown for illustrative purposes:

```
module MyModule
using Lib

using BigLib: thing1, thing2

import Base.show

export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

Note that the style is not to indent the body of the module, since that would typically lead to whole files being indented.

This module defines a type `MyType`, and two functions. Function `foo` and type `MyType` are exported, and so will be available for importing into other modules. Function `bar` is private to `MyModule`.

The statement `using Lib` means that a module called `Lib` will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it among variables exported by `Lib` and import it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in `Lib`.

The statement `using BigLib: thing1, thing2` brings just the identifiers `thing1` and `thing2` into scope from module `BigLib`. If these names refer to functions, adding methods to them will not be allowed (you may only "use" them, not extend them).

The `import` keyword supports the same syntax as `using`. It does not add modules to be searched the way `using` does. `import` also differs from `using` in that functions imported using `import` can be extended with new methods.

In `MyModule` above we wanted to add a method to the standard `show` function, so we had to write `import Base.show`. Functions whose names are only visible via `using` cannot be extended.

Once a variable is made visible via `using` or `import`, a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

16.1 Summary of module usage

To load a module, two main keywords can be used: `using` and `import`. To understand their differences, consider the following example:

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

In this module we export the `x` and `y` functions (with the keyword `export`), and also have the non-exported function `p`. There are several different ways to load the Module and its inner functions into the current workspace:

Import Command	What is brought into scope	Available for method extension
`using MyModule`	All exported names (`x` and `y`), `MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`using MyModule: x, p`	`x` and `p`	
`import MyModule`	`MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`import MyModule.x, MyModule.p`	`x` and `p`	`x` and `p`
`import MyModule: x, p`	`x` and `p`	`x` and `p`

Modules and files

Files and file names are mostly unrelated to modules; modules are associated only with module expressions. One can have multiple files per module, and multiple modules per file:

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

Including the same code in different modules provides mixin-like behavior. One could use this to run the same code with different base definitions, for example testing code by running it with "safe" versions of some operators:

```
module Normal
include("mycode.jl")
end

module Testing
include("safe_operators.jl")
include("mycode.jl")
end
```

Standard modules

There are three important standard modules:

- `Core` contains all functionality "built into" the language.
- `Base` contains basic functionality that is useful in almost all cases.
- `Main` is the top-level module and the current module, when Julia is started.

Default top-level definitions and bare modules

In addition to `using Base`, modules also automatically contain definitions of the `eval` and `include` functions, which evaluate expressions/files within the global scope of that module.

If these default definitions are not wanted, modules can be defined using the keyword `baremodule` instead (note: `Core` is still imported, as per above). In terms of `baremodule`, a standard `module` looks like this:

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

Relative and absolute module paths

Given the statement `using Foo`, the system consults an internal table of top-level modules to look for one named `Foo`. If the module does not exist, the system attempts to `require(:Foo)`, which typically results in loading code from an installed package.

However, some modules contain submodules, which means you sometimes need to access a non-top-level module. There are two ways to do this. The first is to use an absolute path, for example `using Base.Sort`. The second is to use a relative path, which makes it easier to import submodules of the current module or any of its enclosing modules:

```
module Parent

module Utils
...
end

using .Utils
```

```
...
end
```

Here module `Parent` contains a submodule `Utils`, and code in `Parent` wants the contents of `Utils` to be visible. This is done by starting the `using` path with a period. Adding more leading periods moves up additional levels in the module hierarchy. For example `using ..Utils` would look for `Utils` in `Parent`'s enclosing module rather than in `Parent` itself.

Note that relative-import qualifiers are only valid in `using` and `import` statements.

Namespace miscellanea

If a name is qualified (e.g. `Base.sin`), then it can be accessed even if it is not exported. This is often useful when debugging. It can also have methods added to it by using the qualified name as the function name. However, due to syntactic ambiguities that arise, if you wish to add methods to a function in a different module whose name contains only symbols, such as an operator, `Base.+` for example, you must use `Base.:+` to refer to it. If the operator is more than one character in length you must surround it in brackets, such as: `Base.:(==)`.

Macro names are written with `@` in import and export statements, e.g. `import Mod.@mac`. Macros in other modules can be invoked as `Mod.@mac` or `@Mod.mac`.

The syntax `M.x = y` does not work to assign a global in another module; global assignment is always module-local.

A variable name can be "reserved" without assigning to it by declaring it as `global x`. This prevents name conflicts for globals initialized after load time.

Module initialization and precompilation

Large modules can take several seconds to load because executing all of the statements in a module often involves compiling a large amount of code. Julia creates precompiled caches of the module to reduce this time.

The incremental precompiled module file are created and used automatically when using `import` or `using` to load a module. This will cause it to be automatically compiled the first time it is imported. Alternatively, you can manually call `Base.compilecache(modulename)`. The resulting cache files will be stored in `DEPOT_PATH[1]/compiled/`. Subsequently, the module is automatically recompiled upon `using` or `import` whenever any of its dependencies change; dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by `include_dependency(path)` in the module file(s).

For file dependencies, a change is determined by examining whether the modification time (mtime) of each file loaded by `include` or added explicitly by `include_dependency` is unchanged, or equal to the modification time truncated to the nearest second (to accommodate systems that can't copy mtime with sub-second accuracy). It also takes into account whether the path to the file chosen by the search logic in `require` matches the path that had created the

precompile file. It also takes into account the set of dependencies already loaded into the current process and won't recompile those modules, even if their files change or disappear, in order to avoid creating incompatibilities between the running system and the precompile cache.

If you know that a module is not safe to precompile your module (for example, for one of the reasons described below), you should put `__precompile__(false)` in the module file (typically placed at the top). This will cause `Base.compilecache` to throw an error, and will cause `using` / `import` to load it directly into the current process and skip the precompile and caching. This also thereby prevents the module from being imported by any other precompiled module.

You may need to be aware of certain behaviors inherent in the creation of incremental shared libraries which may require care when writing your module. For example, external state is not preserved. To accommodate this, explicitly separate any initialization steps that must occur at runtime from steps that can occur at compile time. For this purpose, Julia allows you to define an `__init__()` function in your module that executes any initialization steps that must occur at runtime. This function will not be called during compilation (`--output-*`). Effectively, you can assume it will be run exactly once in the lifetime of the code. You may, of course, call it manually if necessary, but the default is to assume this function deals with computing state for the local machine, which does not need to be – or even should not be – captured in the compiled image. It will be called after the module is loaded into a process, including if it is being loaded into an incremental compile (`--output-incremental=yes`), but not if it is being loaded into a full-compilation process.

In particular, if you define a `function __init__()` in a module, then Julia will call `__init__()` immediately after the module is loaded (e.g., by `import`, `using`, or `require`) at runtime for the first time (i.e., `__init__` is only called once, and only after all statements in the module have been executed). Because it is called after the module is fully imported, any submodules or other imported modules have their `__init__` functions called before the `__init__` of the enclosing module.

Two typical uses of `__init__` are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. For example, suppose that we are calling a C library `libfoo` that requires us to call a `foo_init()` initialization function at runtime. Suppose that we also want to define a global constant `foo_data_ptr` that holds the return value of a `void *foo_data()` function defined by `libfoo` – this constant must be initialized at runtime (not at compile time) because the pointer address will change from run to run. You could accomplish this by defining the following `__init__` function in your module:

```
const foo_data_ptr = Ref{Ptr{Cvoid}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Cvoid, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Cvoid}, ())
    nothing
end
```

Notice that it is perfectly possible to define a global inside a function like `__init__`; this is one of the advantages of using a dynamic language. But by making it a constant at global scope, we can ensure that the type is known to the compiler and allow it to generate better optimized code. Obviously, any other globals in your module that depends on `foo_data_ptr` would also have to be initialized in `__init__`.

Constants involving most Julia objects that are not produced by `ccall` do not need to be placed in `__init__`: their definitions can be precompiled and loaded from the cached module image. This includes complicated heap-allocated objects like arrays. However, any routine that returns a raw pointer value must be called at runtime for precompilation to work (`Ptr` objects will turn into null pointers unless they are hidden inside an `isbits` object). This includes the return values of the Julia functions `cfunction` and `pointer`.

Dictionary and set types, or in general anything that depends on the output of a `hash(key)` method, are a trickier case. In the common case where the keys are numbers, strings, symbols, ranges, `Expr`, or compositions of these types (via arrays, tuples, sets, pairs, etc.) they are safe to precompile. However, for a few other key types, such as `Function` or `DataType` and generic user-defined types where you haven't defined a `hash` method, the fallback `hash` method depends on the memory address of the object (via its `objectid`) and hence may change from run to run. If you have one of these key types, or if you aren't sure, to be safe you can initialize this dictionary from within your `__init__` function. Alternatively, you can use the `IdDict` dictionary type, which is specially handled by precompilation so that it is safe to initialize at compile-time.

When using precompilation, it is important to keep a clear sense of the distinction between the compilation phase and the execution phase. In this mode, it will often be much more clearly apparent that Julia is a compiler which allows execution of arbitrary Julia code, not a standalone interpreter that also generates compiled code.

Other known potential failure scenarios include:

1. Global counters (for example, for attempting to uniquely identify objects). Consider the following code snippet:

   ```
   mutable struct UniquedById
       myid::Int
       let counter = 0
           UniquedById() = new(counter += 1)
       end
   end
   ```

   while the intent of this code was to give every instance a unique id, the counter value is recorded at the end of compilation. All subsequent usages of this incrementally compiled module will start from that same counter value.

   Note that `objectid` (which works by hashing the memory pointer) has similar issues (see notes on `Dict` usage below).

One alternative is to use a macro to capture `@__MODULE__` and store it alone with the current `counter` value, however, it may be better to redesign the code to not depend on this global state.

2. Associative collections (such as `Dict` and `Set`) need to be re-hashed in `__init__`. (In the future, a mechanism may be provided to register an initializer function.)

3. Depending on compile-time side-effects persisting through load-time. Example include: modifying arrays or other variables in other Julia modules; maintaining handles to open files or devices; storing pointers to other system resources (including memory);

4. Creating accidental "copies" of global state from another module, by referencing it directly instead of via its lookup path. For example, (in global scope):

```
#mystdout = Base.stdout #= will not work correctly, since this will copy Base.stdout into this module =#
# instead use accessor functions:
getstdout() = Base.stdout #= best option =#
# or move the assignment into the runtime:
__init__() = global mystdout = Base.stdout #= also works =#
```

Several additional restrictions are placed on the operations that can be done while precompiling code to help the user avoid other wrong-behavior situations:

1. Calling `eval` to cause a side-effect in another module. This will also cause a warning to be emitted when the incremental precompile flag is set.

2. `global const` statements from local scope after `__init__()` has been started (see issue #12010 for plans to add an error for this)

3. Replacing a module is a runtime error while doing an incremental precompile.

A few other points to be aware of:

1. No code reload / cache invalidation is performed after changes are made to the source files themselves, (including by `Pkg.update`), and no cleanup is done after `Pkg.rm`

2. The memory sharing behavior of a reshaped array is disregarded by precompilation (each view gets its own copy)

3. Expecting the filesystem to be unchanged between compile-time and runtime e.g. `@__FILE__`/`source_path()` to find resources at runtime, or the BinDeps `@checked_lib` macro. Sometimes this is unavoidable. However, when possible, it can be good practice to copy resources into the module at compile-time so they won't need to be found at runtime.

4. `WeakRef` objects and finalizers are not currently handled properly by the serializer (this will be fixed in an upcoming release).

5. It is usually best to avoid capturing references to instances of internal metadata objects such as `Method`, `MethodInstance`, `MethodTable`, `TypeMapLevel`, `TypeMapEntry` and fields of those objects, as this can confuse the serializer and may not lead to the outcome you desire. It is not necessarily an error to do this, but you simply need to be prepared that the system will try to copy some of these and to create a single unique instance of others.

It is sometimes helpful during module development to turn off incremental precompilation. The command line flag `--compiled-modules={yes|no}` enables you to toggle module precompilation on and off. When Julia is started with `--compiled-modules=no` the serialized modules in the compile cache are ignored when loading modules and module dependencies. `Base.compilecache` can still be called manually. The state of this command line flag is passed to `Pkg.build` to disable automatic precompilation triggering when installing, updating, and explicitly building packages.

Chapter 17

Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system since Julia 0.4.

The basic syntax is simple: any string appearing at the top-level right before an object (function, macro, type or instance) will be interpreted as documenting it (these are called docstrings). Note that no blank lines or comments may intervene between a docstring and the documented object. Here is a basic example:

```
"Tell whether there are too foo items in the array."
foo(xs::Array) = ...
```

Documentation is interpreted as Markdown, so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the `@doc` macro just as well.

> Note
>
> Markdown support is implemented in the `Markdown` standard library and for a full list of supported syntax see the documentation.

Here is a more complex example, still using Markdown:

```
"""
    bar(x[, y])

Compute the Bar index between `x` and `y`. If `y` is missing, compute
the Bar index between all pairs of columns of `x`.
```

```
# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
```
"""
function bar(x, y) ...
```

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

   This can be identical to the signature present in the Julia code (like `mean(x::AbstractArray)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `f(x, y=1)`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `f(x[, y])` and `f(x[, y[, z]])`). An alternative solution is to use several lines: one without optional arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a `<keyword arguments>` placeholder in the signature (i.e. `f(x; <keyword arguments>)`), and give the complete list under an `# Arguments` section (see point 4 below).

2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

   The one-line sentence should use the imperative form ("Do this", "Return that") instead of the third person (do not write "Returns the length...") when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).

3. Do not repeat yourself.

   Since the function name is given by the signature, there is no need to start the documentation with "The function `bar`...": go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.

4. Only provide an argument list when really necessary.

   For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function's purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular

keyword arguments). In that case, insert it after the general description of the function, under an `# Arguments` header, with one `-` bullet for each argument. The list should mention the types and default values (if any) of the arguments:

```
"""
...
# Arguments
- `n::Integer`: the number of elements to compute.
- `dim::Integer=1`: the dimensions along which to perform the computation.
...
"""
```

5. Provide hints to related functions.

   Sometimes there are functions of related functionality. To increase discoverability please provide a short list of these in a `See also:` paragraph.

   ```
   See also: [`bar!`](@ref), [`baz`](@ref), [`baaz`](@ref)
   ```

6. Include any code examples in an `# Examples` section.

   Examples should, whenever possible, be written as doctests. A doctest is a fenced code block (see Code blocks) starting with ```` ```jldoctest ```` and contains any number of `julia>` prompts together with inputs and expected outputs that mimic the Julia REPL.

   > Note
   >
   > Doctests are enabled by `Documenter.jl`. For more detailed documentation see Documenter's manual.

   For example in the following docstring a variable `a` is defined and the expected result, as printed in a Julia REPL, appears afterwards:

   ````
   """
   Some nice documentation here.

   # Examples
   ```jldoctest
   julia> a = [1 2; 3 4]
   2×2 Array{Int64,2}:
    1  2
    3  4
   ```
   """
   ````

> Warning
>
> Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly construct and seed your own `MersenneTwister` (or other pseudorandom number generator) and pass it to the functions you are doctesting.
>
> Operating system word size (`Int32` or `Int64`) as well as path separator differences (`/` or `\`) will also affect the reproducibility of some doctests.
>
> Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

You can then run `make -C doc doctest=true` to run all the doctests in the Julia Manual and API documentation, which will ensure that your example works.

To indicate that the output result is truncated, you may write `[...]` at the line where checking should stop. This is useful to hide a stacktrace (which contains non-permanent references to lines of julia code) when the doctest shows that an exception is thrown, for example:

```
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
[...]
```
```

Examples that are untestable should be written within fenced code blocks starting with ```` ```julia ```` so that they are highlighted correctly in the generated documentation.

> Tip
>
> Wherever possible examples should be self-contained and runnable so that readers are able to try them out without having to include any dependencies.

7. Use backticks to identify code and equations.

   Julia identifiers and code excerpts should always appear between backticks `` ` `` to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks ``` `` ```. Use Unicode characters rather than their LaTeX escape sequence, i.e. ``` ``α = 1`` ``` rather than ``` ``\\alpha = 1`` ```.

8. Place the starting and ending """ characters on lines by themselves.

   That is, write:

```
"""
...


...
"""
f(x, y) = ...
```

rather than:

```
"""...


..."""
f(x, y) = ...
```

This makes it more clear where docstrings start and end.

9. Respect the line length limit used in the surrounding code.

   Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It is advised to add line breaks after 92 characters.

10. Provide information allowing custom types to implement the function in an `# Implementation` section. These implementation details intended for developers rather than users, explaining e.g. which functions should be overridden and which functions automatically use appropriate fallbacks, are better kept separate from the main description of the function's behavior.

17.1 Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing `?` followed by the name of a function or macro, and pressing `Enter`. For example,

```
?cos
?@time
?r""
```

will bring up docs for the relevant function, macro or string macro respectively. In Juno using `Ctrl-J, Ctrl-D` will bring up documentation for the object under the cursor.

17.2 Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general,

only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by `function bar end`). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere. For example:

```
"""
    *(x, y, z...)

Multiplication operator. `x * y * z *...` calls this function with multiple
arguments, i.e. `*(x, y, z...)`.
"""
function *(x, y, z...)
    # ... [implementation sold separately] ...
end

"""
    *(x::AbstractString, y::AbstractString, z::AbstractString...)

When applied to strings, concatenates them.
"""
function *(x::AbstractString, y::AbstractString, z::AbstractString...)
    # ... [insert secret sauce here] ...
end

help?> *
search: * .*

  *(x, y, z...)

  Multiplication operator. x * y * z *... calls this function with multiple
  arguments, i.e. *(x,y,z...).

  *(x::AbstractString, y::AbstractString, z::AbstractString...)

  When applied to strings, concatenates them.
```

When retrieving documentation for a generic function, the metadata for each method is concatenated with the `catdoc` function, which can of course be overridden for custom types.

17.3 Advanced Usage

The `@doc` macro associates its first argument with its second in a per-module dictionary called `META`.

To make it easier to write documentation, the parser treats the macro name `@doc` specially: if a call to `@doc` has one argument, but another expression appears after a single line break, then that additional expression is added as an argument to the macro. Therefore the following syntax is parsed as a 2-argument call to `@doc`:

```
@doc raw"""
...
"""
f(x) = x
```

This makes it possible to use expressions other than normal string literals (such as the `raw""` string macro) as a docstring.

When used for retrieving documentation, the `@doc` macro (or equally, the `doc` function) will search all `META` dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example) will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

```
@doc "..." foo!
@doc (@doc foo!) foo
```

Or for use with Julia's metaprogramming functionality:

```
for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*), (:divide, :/))
    @eval begin
        $f(a,b) = $op(a,b)
    end
end
@doc "`add(a,b)` adds `a` and `b` together" add
@doc "`subtract(a,b)` subtracts `b` from `a`" subtract
```

Documentation written in non-toplevel blocks, such as `begin`, `if`, `for`, and `let`, is added to the documentation system as blocks are evaluated. For example:

```
if condition()
    "..."
```

```
    f(x) = x
end
```

will add documentation to `f(x)` when `condition()` is `true`. Note that even if `f(x)` goes out of scope at the end of the block, its documentation will remain.

Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to `Docs.getdoc` for your custom type that returns the documentation on a per-instance basis. For instance,

```
struct MyType
    value::String
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType("x")
y = MyType("y")
```

`?x` will display "Documentation for MyType with value x" while `?y` will display "Documentation for MyType with value y".

17.4 Syntax Guide

A comprehensive overview of all documentable Julia syntax. In the following examples "`...`" is used to illustrate an arbitrary docstring.

$ and \ characters

The $ and \ characters are still parsed as string interpolation or start of an escape sequence in docstrings too. The `raw""` string macro together with the `@doc` macro can be used to avoid having to escape them. This is handy when the docstrings include LaTeX or Julia source code examples containing interpolation:

````
@doc raw"""
```math
\LaTeX
```
````

```
"""
function f end
```

Functions and Methods

```
"..."
function f end

"..."
f
```

Adds docstring `"..."` to the function `f`. The first version is the preferred syntax, however both are equivalent.

```
"..."
f(x) = x

"..."
function f(x)
    x
end

"..."
f(x)
```

Adds docstring `"..."` to the method `f(::Any)`.

```
"..."
f(x, y = 1) = x + y
```

Adds docstring `"..."` to two `Methods`, namely `f(::Any)` and `f(::Any, ::Any)`.

Macros

```
"..."
macro m(x) end
```

Adds docstring `"..."` to the `@m(::Any)` macro definition.

```
"..."
:(@m)
```

Adds docstring `"..."` to the macro named `@m`.

Types

```
"..."
abstract type T1 end

"..."
mutable struct T2
    ...
end

"..."
struct T3
    ...
end
```

Adds the docstring `"..."` to types `T1`, `T2`, and `T3`.

```
"..."
struct T
    "x"
    x
    "y"
    y
end
```

Adds docstring `"..."` to type `T`, `"x"` to field `T.x` and `"y"` to field `T.y`. Also applicable to `mutable struct` types.

Modules

```
"..."
module M end

module M

"..."
M

end
```

Adds docstring `"..."` to the `ModuleM`. Adding the docstring above the `Module` is the preferred syntax, however both are equivalent.

```
"..."
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

Documenting a `baremodule` by placing a docstring above the expression automatically imports `@doc` into the module. These imports must be done manually when the module expression is not documented. Empty `baremodule`s cannot be documented.

Global Variables

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring `"..."` to the `Binding`s `a`, `b`, and `c`.

`Binding`s are used to store a reference to a particular `Symbol` in a `Module` without storing the referenced value itself.

> Note
>
> When a `const` definition is only used to define an alias of another definition, such as is the case with the function `div` and its alias `÷` in `Base`, do not document the alias and instead document the actual function.

If the alias is documented and not the real definition then the docsystem (`?` mode) will not return the docstring attached to the alias when the real definition is searched for.

For example you should write

```
"..."
f(x) = x + 1
const alias = f
```

rather than

```
f(x) = x + 1
"..."
const alias = f
```

```
"..."
sym
```

Adds docstring `"..."` to the value associated with `sym`. Users should prefer documenting `sym` at its definition.

Multiple Objects

```
"..."
a, b
```

Adds docstring `"..."` to `a` and `b` each of which should be a documentable expression. This syntax is equivalent to

```
"..."
a

"..."
b
```

Any number of expressions many be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions `f` and `f!`.

Macro-generated code

```
"..."
@m expression
```

Adds docstring "..." to expression generated by expanding `@m expression`. This allows for expressions decorated with `@inline`, `@noinline`, `@generated`, or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the `@__doc__` macro.

The `@enum` macro makes use of `@__doc__` to allow for documenting `Enum`s. Examining its definition should serve as an example of how to use `@__doc__` correctly.

`Core.@__doc__` – Macro.

```
@__doc__(ex)
```

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @__doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

source

Chapter 18

Metaprogramming

The strongest legacy of Lisp in the Julia language is its metaprogramming support. Like Lisp, Julia represents its own code as a data structure of the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of abstract syntax trees. In contrast, preprocessor "macro" systems, like that of C and C++, perform textual manipulation and substitution before any actual parsing or interpretation occurs. Because all data types and code in Julia are represented by Julia data structures, powerful reflection capabilities are available to explore the internals of a program and its types just like any other data.

18.1 Program representation

Every Julia program starts life as a string:

```
julia> prog = "1 + 1"
"1 + 1"
```

What happens next?

The next step is to parse each string into an object called an expression, represented by the Julia type `Expr`:

```
julia> ex1 = Meta.parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

`Expr` objects contain two parts:

- a `Symbol` identifying the kind of expression. A symbol is an interned string identifier (more discussion below).

```
julia> ex1.head
:call
```

- the expression arguments, which may be symbols, other expressions, or literal values:

```
julia> ex1.args
3-element Array{Any,1}:
  :+
 1
 1
```

Expressions may also be constructed directly in prefix notation:

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

The two expressions constructed above – by parsing and by direct construction – are equivalent:

```
julia> ex1 == ex2
true
```

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

The `dump` function provides indented and annotated display of `Expr` objects:

```
julia> dump(ex2)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
```

`Expr` objects may also be nested:

```
julia> ex3 = Meta.parse("(4 + 4) / 2")
:((4 + 4) / 2)
```

Another way to view expressions is with `Meta.show_sexpr`, which displays the S-expression form of a given `Expr`, which may look very familiar to users of Lisp. Here's an example illustrating the display on a nested `Expr`:

```
julia> Meta.show_sexpr(ex3)
(:call, :/, (:call, :+, 4, 4), 2)
```

Symbols

The `:` character has two syntactic purposes in Julia. The first form creates a `Symbol`, an interned string used as one building-block of expressions:

```
julia> :foo
:foo

julia> typeof(ans)
Symbol
```

The `Symbol` constructor takes any number of arguments and creates a new symbol by concatenating their string representations together:

```
julia> :foo == Symbol("foo")
true

julia> Symbol("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

Note that to use `:` syntax, the symbol's name must be a valid identifier. Otherwise the `Symbol(str)` constructor must be used.

In the context of an expression, symbols are used to indicate access to variables; when an expression is evaluated, a symbol is replaced with the value bound to that symbol in the appropriate scope.

Sometimes extra parentheses around the argument to `:` are needed to avoid ambiguity in parsing:

```
julia> :(:)
:(:)

julia> :(::)
:(::)
```

18.2 Expressions and evaluation

Quoting

The second syntactic purpose of the `:` character is to create expression objects without using the explicit `Expr` constructor. This is referred to as quoting. The `:` character, followed by paired parentheses around a single statement of Julia code, produces an `Expr` object based on the enclosed code. Here is example of the short form used to quote an arithmetic expression:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr
```

(to view the structure of this expression, try `ex.head` and `ex.args`, or use `dump` as above or `Meta.@dump`)

Note that equivalent expressions may be constructed using `Meta.parse` or the direct `Expr` form:

```
julia>      :(a + b*c + 1)       ==
       Meta.parse("a + b*c + 1") ==
       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can have arbitrary run-time values without literal forms as args. In this specific example, `+` and `a` are symbols, `*(b,c)` is a subexpression, and `1` is a literal 64-bit signed integer.

There is a second syntactic form of quoting for multiple expressions: blocks of code enclosed in `quote ... end`.

```
julia> ex = quote
           x = 1
           y = 2
           x + y
```

```
        end
quote
    #= none:2 =#
    x = 1
    #= none:3 =#
    y = 2
    #= none:4 =#
    x + y
end

julia> typeof(ex)
Expr
```

Interpolation

Direct construction of `Expr` objects with value arguments is powerful, but `Expr` constructors can be tedious compared to "normal" Julia syntax. As an alternative, Julia allows interpolation of literals or expressions into quoted expressions. Interpolation is indicated by a prefix `$`.

In this example, the value of variable `a` is interpolated:

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

```
julia> $a + b
ERROR: syntax: "$" expression outside quote
```

In this example, the tuple `(1,2,3)` is interpolated as an expression into a conditional test:

```
julia> ex = :(a in $:((1,2,3)) )
:(a in (1, 2, 3))
```

The use of `$` for expression interpolation is intentionally reminiscent of string interpolation and command interpolation. Expression interpolation allows convenient, readable programmatic construction of complex Julia expressions.

Splatting interpolation

Notice that the `$` interpolation syntax allows inserting only a single expression into an enclosing expression. Occasionally, you have an array of expressions and need them all to become arguments of the surrounding expression. This can be done with the syntax `$(xs...)`. For example, the following code generates a function call where the number of arguments is determined programmatically:

```
julia> args = [:x, :y, :z];

julia> :(f(1, $(args...)))
:(f(1, x, y, z))
```

Nested quote

Naturally, it is possible for quote expressions to contain other quote expressions. Understanding how interpolation works in these cases can be a bit tricky. Consider this example:

```
julia> x = :(1 + 2);

julia> e = quote quote $x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
    #= none:1 =#
    $(Expr(:$, :x))
end))
end
```

Notice that the result contains `Expr(:$, :x)`, which means that `x` has not been evaluated yet. In other words, the `$` expression "belongs to" the inner quote expression, and so its argument is only evaluated when the inner quote expression is:

```
julia> eval(e)
quote
    #= none:1 =#
    1 + 2
end
```

However, the outer `quote` expression is able to interpolate values inside the `$` in the inner quote. This is done with multiple `$`s:

```
julia> e = quote quote $$x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
    #= none:1 =#
    $(Expr(:$, :(1 + 2)))
end))
end
```

Notice that `:(1 + 2)` now appears in the result instead of the symbol `:x`. Evaluating this expression yields an interpolated `3`:

```
julia> eval(e)
quote
    #= none:1 =#
    3
end
```

The intuition behind this behavior is that `x` is evaluated once for each `$`: one `$` works similarly to `eval(:x)`, giving `x`'s value, while two `$`s do the equivalent of `eval(eval(:x))`.

QuoteNode

The usual representation of a `quote` form in an AST is an `Expr` with head `:quote`:

```
julia> dump(Meta.parse(":(1+2)"))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 1
        3: Int64 2
```

As we have seen, such expressions support interpolation with `$`. However, in some situations it is necessary to quote code without performing interpolation. This kind of quoting does not yet have syntax, but is represented internally as an object of type `QuoteNode`:

```
julia> eval(Meta.quot(Expr(:$, :(1+2))))
3

julia> eval(QuoteNode(Expr(:$, :(1+2))))
:($(Expr(:$, :(1 + 2))))
```

The parser yields `QuoteNode`s for simple quoted items like symbols:

```
julia> dump(Meta.parse(":x"))
QuoteNode
  value: Symbol x
```

`QuoteNode` can also be used for certain advanced metaprogramming tasks.

`eval` and effects

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using `eval`:

```
julia> :(1 + 2)
:(1 + 2)

julia> eval(ans)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: UndefVarError: b not defined
[...]

julia> a = 1; b = 2;

julia> eval(ex)
3
```

Every module has its own `eval` function that evaluates expressions in its global scope. Expressions passed to `eval` are not limited to returning values – they can also have side-effects that alter the state of the enclosing module's environment:

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: x not defined

julia> eval(ex)
1

julia> x
1
```

Here, the evaluation of an expression object causes a value to be assigned to the global variable `x`.

Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, it is possible to dynamically generate arbitrary code which can then be run using `eval`. Here is a simple example:

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)

julia> a = 0; b = 2;

julia> eval(ex)
3
```

The value of `a` is used to construct the expression `ex` which applies the `+` function to the value 1 and the variable `b`. Note the important distinction between the way `a` and `b` are used:

- The value of the variable `a` at expression construction time is used as an immediate value in the expression. Thus, the value of `a` when the expression is evaluated no longer matters: the value in the expression is already `1`, independent of whatever the value of `a` might be.
- On the other hand, the symbol `:b` is used in the expression construction, so the value of the variable `b` at that time is irrelevant – `:b` is just a symbol and the variable `b` need not even be defined. At expression evaluation time, however, the value of the symbol `:b` is resolved by looking up the value of the variable `b`.

Functions on Expressions

As hinted above, one extremely useful feature of Julia is the capability to generate and manipulate Julia code within Julia itself. We have already seen one example of a function returning `Expr` objects: the `parse` function, which takes a string of Julia code and returns the corresponding `Expr`. A function can also take one or more `Expr` objects as arguments, and return another `Expr`. Here is a simple, motivating example:

```
julia> function math_expr(op, op1, op2)
           expr = Expr(:call, op, op1, op2)
           return expr
       end
math_expr (generic function with 1 method)

julia>  ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))
:(1 + 4 * 5)

julia> eval(ex)
21
```

As another example, here is a function that doubles any numeric argument, but leaves expressions alone:

```
julia> function make_expr2(op, opr1, opr2)
           opr1f, opr2f = map(x -> isa(x, Number) ? 2*x : x, (opr1, opr2))
           retexpr = Expr(:call, op, opr1f, opr2f)
           return retexpr
       end
make_expr2 (generic function with 1 method)

julia> make_expr2(:+, 1, 2)
:(2 + 4)

julia> ex = make_expr2(:+, 1, Expr(:call, :*, 5, 8))
:(2 + 5 * 8)

julia> eval(ex)
42
```

18.3 Macros

Macros provide a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned expression, and the resulting expression is compiled directly rather than requiring a runtime `eval` call. Macro arguments may include expressions, literal values, and symbols.

Basics

Here is an extraordinarily simple macro:

```
julia> macro sayhello()
           return :( println("Hello, world!") )
       end
@sayhello (macro with 1 method)
```

Macros have a dedicated character in Julia's syntax: the `@` (at-sign), followed by the unique name declared in a `macro NAME ... end` block. In this example, the compiler will replace all instances of `@sayhello` with:

```
:( println("Hello, world!") )
```

When `@sayhello` is entered in the REPL, the expression executes immediately, thus we only see the evaluation result:

```
julia> @sayhello()
Hello, world!
```

Now, consider a slightly more complex macro:

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name) )
       end
@sayhello (macro with 1 method)
```

This macro takes one argument: `name`. When `@sayhello` is encountered, the quoted expression is expanded to interpolate the value of the argument into the final expression:

```
julia> @sayhello("human")
Hello, human
```

We can view the quoted return expression using the function `macroexpand` (important note: this is an extremely useful tool for debugging macros):

```
julia> ex = macroexpand(Main, :(@sayhello("human")) )
:(Main.println("Hello, ", "human"))

julia> typeof(ex)
Expr
```

We can see that the `"human"` literal has been interpolated into the expression.

There also exists a macro `@macroexpand` that is perhaps a bit more convenient than the `macroexpand` function:

```
julia> @macroexpand @sayhello "human"
:(println("Hello, ", "human"))
```

Hold up: why macros?

We have already seen a function `f(::Expr...) -> Expr` in a previous section. In fact, `macroexpand` is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code before the full program is run. To illustrate the difference, consider the following example:

```
julia> macro twostep(arg)
           println("I execute at parse time. The argument is: ", arg)
           return :(println("I execute at runtime. The argument is: ", $arg))
       end
@twostep (macro with 1 method)

julia> ex = macroexpand(Main, :(@twostep :(1, 2, 3)) );
I execute at parse time. The argument is: $(Expr(:quote, :((1, 2, 3))))
```

The first call to `println` is executed when `macroexpand` is called. The resulting expression contains only the second `println`:

```
julia> typeof(ex)
Expr

julia> ex
:(println("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:((1, 2, 3))))))))
```

```
julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

Macro invocation

Macros are invoked with the following general syntax:

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

Note the distinguishing `@` before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after `@name` in the second form. The two styles should not be mixed. For example, the following syntax is different from the examples above; it passes the tuple `(expr1, expr2, ...)` as one argument to the macro:

```
@name (expr1, expr2, ...)
```

An alternative way to invoke a macro over an array literal (or comprehension) is to juxtapose both without using parentheses. In this case, the array will be the only expression fed to the macro. The following syntax is equivalent (and different from `@name [a b] * v`):

```
@name[a b] * v
@name([a b]) * v
```

It is important to emphasize that macros receive their arguments as expressions, literals, or symbols. One way to explore macro arguments is to call the `show` function within the macro body:

```
julia> macro showarg(x)
           show(x)
           # ... remainder of macro, returning an expression
       end
@showarg (macro with 1 method)

julia> @showarg(a)
:a

julia> @showarg(1+1)
```

```
:(1 + 1)

julia> @showarg(println("Yo!"))
:(println("Yo!"))
```

In addition to the given argument list, every macro is passed extra arguments named `__source__` and `__module__`.

The argument `__source__` provides information (in the form of a `LineNumberNode` object) about the parser location of the `@` sign from the macro invocation. This allows macros to include better error diagnostic information, and is commonly used by logging, string-parser macros, and docs, for example, as well as to implement the `@__LINE__`, `@__FILE__`, and `@__DIR__` macros.

The location information can be accessed by referencing `__source__.line` and `__source__.file`:

```
julia> macro __LOCATION__(); return QuoteNode(__source__); end
@__LOCATION__ (macro with 1 method)

julia> dump(
           @__LOCATION__(
       ))
LineNumberNode
  line: Int64 2
  file: Symbol none
```

The argument `__module__` provides information (in the form of a `Module` object) about the expansion context of the macro invocation. This allows macros to look up contextual information, such as existing bindings, or to insert the value as an extra argument to a runtime function call doing self-reflection in the current module.

Building an advanced macro

Here is a simplified definition of Julia's `@assert` macro:

```
julia> macro assert(ex)
           return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
       end
@assert (macro with 1 method)
```

This macro can be used like this:

```
julia> @assert 1 == 1.0

julia> @assert 1 == 0
ERROR: AssertionError: 1 == 0
```

In place of the written syntax, the macro call is expanded at parse time to its returned result. This is equivalent to writing:

```
1 == 1.0 ? nothing : throw(AssertionError("1 == 1.0"))
1 == 0 ? nothing : throw(AssertionError("1 == 0"))
```

That is, in the first call, the expression `:(1 == 1.0)` is spliced into the test condition slot, while the value of `string(:(1 == 1.0))` is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the `@assert` macro call occurs. Then at execution time, if the test expression evaluates to true, then `nothing` is returned, whereas if the test is false, an error is raised indicating the asserted expression that was false. Notice that it would not be possible to write this as a function, since only the value of the condition is available and it would be impossible to display the expression that computed it in the error message.

The actual definition of `@assert` in Julia Base is more complicated. It allows the user to optionally specify their own error message, instead of just printing the failed expression. Just like in functions with a variable number of arguments (가변인자 함수), this is specified with an ellipses following the last argument:

```
julia> macro assert(ex, msgs...)
           msg_body = isempty(msgs) ? ex : msgs[1]
           msg = string(msg_body)
           return :($ex ? nothing : throw(AssertionError($msg)))
       end
@assert (macro with 1 method)
```

Now `@assert` has two modes of operation, depending upon the number of arguments it receives! If there's only one argument, the tuple of expressions captured by `msgs` will be empty and it will behave the same as the simpler definition above. But now if the user specifies a second argument, it is printed in the message body instead of the failing expression. You can inspect the result of a macro expansion with the aptly named `@macroexpand` macro:

```
julia> @macroexpand @assert a == b
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a == b"))
```

```
    end)

julia> @macroexpand @assert a==b "a should equal b!"
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a should equal b!"))
    end)
```

There is yet another case that the actual `@assert` macro handles: what if, in addition to printing "a should equal b," we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., `@assert a==b "a ($a) should equal b ($b)!"`, but this won't work as expected with the above macro. Can you see why? Recall from string interpolation that an interpolated string is rewritten to a call to `string`. Compare:

```
julia> typeof(:("a should equal b"))
String

julia> typeof(:("a ($a) should equal b ($b)!"))
Expr

julia> dump(:("a ($a) should equal b ($b)!"))
Expr
  head: Symbol string
  args: Array{Any}((5,))
    1: String "a ("
    2: Symbol a
    3: String ") should equal b ("
    4: Symbol b
    5: String ")!"
```

So now instead of getting a plain string in `msg_body`, the macro is receiving a full expression that will need to be evaluated in order to display as expected. This can be spliced directly into the returned expression as an argument to the `string` call; see `error.jl` for the complete implementation.

The `@assert` macro makes great use of splicing into quoted expressions to simplify the manipulation of expressions inside the macro body.

Hygiene

An issue that arises in more complex macros is that of hygiene. In short, macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. Conversely, the expressions that are passed into a macro as arguments are often expected to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module. Julia already has a major advantage over languages with textual macro expansion (like C) in that it only needs to consider the returned expression. All the other variables (such as `msg` in `@assert` above) follow the normal scoping block behavior.

To demonstrate these issues, let us consider writing a `@time` macro that takes an expression as its argument, records the time, evaluates the expression, records the time again, prints the difference between the before and after times, and then has the value of the expression as its final value. The macro might look like this:

```
macro time(ex)
    return quote
        local t0 = time()
        local val = $ex
        local t1 = time()
        println("elapsed time: ", t1-t0, " seconds")
        val
    end
end
```

Here, we want `t0`, `t1`, and `val` to be private temporary variables, and we want `time` to refer to the `time` function in Julia Base, not to any `time` variable the user might have (the same applies to `println`). Imagine the problems that could occur if the user expression `ex` also contained assignments to a variable called `t0`, or defined its own `time` variable. We might get errors, or mysteriously incorrect behavior.

Julia's macro expander solves these problems in the following way. First, variables within a macro result are classified as either local or global. A variable is considered local if it is assigned to (and not declared global), declared local, or used as a function argument name. Otherwise, it is considered global. Local variables are then renamed to be unique (using the `gensym` function, which generates new symbols), and global variables are resolved within the macro definition environment. Therefore both of the above concerns are handled; the macro's locals will not conflict with any user variables, and `time` and `println` will refer to the Julia Base definitions.

One problem remains however. Consider the following use of this macro:

```
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

Here the user expression `ex` is a call to `time`, but not the same `time` function that the macro uses. It clearly refers to `MyModule.time`. Therefore we must arrange for the code in `ex` to be resolved in the macro call environment. This is done by "escaping" the expression with `esc`:

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

An expression wrapped in this manner is left alone by the macro expander and simply pasted into the output verbatim. Therefore it will be resolved in the macro call environment.

This escaping mechanism can be used to "violate" hygiene when necessary, in order to introduce or manipulate user variables. For example, the following macro sets `x` to zero in the call environment:

```
julia> macro zerox()
           return esc(:(x = 0))
       end
@zerox (macro with 1 method)

julia> function foo()
           x = 1
           @zerox
           return x # is zero
       end
foo (generic function with 1 method)

julia> foo()
0
```

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

Getting the hygiene rules correct can be a formidable challenge. Before using a macro, you might want to consider whether a function closure would be sufficient. Another useful strategy is to defer as much work as possible to runtime. For example, many macros simply wrap their arguments in a `QuoteNode` or other similar `Expr`. Some examples of this include `@task body` which simply returns `schedule(Task(() -> $body))`, and `@eval expr`, which simply returns `eval(QuoteNode(expr))`.

To demonstrate, we might rewrite the `@time` example above as:

```
macro time(expr)
    return :(timeit(() -> $(esc(expr))))
end
function timeit(f)
    t0 = time()
    val = f()
    t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    return val
end
```

However, we don't do this for a good reason: wrapping the `expr` in a new scope block (the anonymous function) also slightly changes the meaning of the expression (the scope of any variables in it), while we want `@time` to be usable with minimum impact on the wrapped code.

Macros and dispatch

Macros, just like Julia functions, are generic. This means they can also have multiple method definitions, thanks to multiple dispatch:

```
julia> macro m end
@m (macro with 0 methods)

julia> macro m(args...)
           println("$(length(args)) arguments")
       end
@m (macro with 1 method)

julia> macro m(x,y)
           println("Two arguments")
```

```
       end
@m (macro with 2 methods)

julia> @m "asd"
1 arguments

julia> @m 1 2
Two arguments
```

However one should keep in mind, that macro dispatch is based on the types of AST that are handed to the macro, not the types that the AST evaluates to at runtime:

```
julia> macro m(::Int)
           println("An Integer")
       end
@m (macro with 3 methods)

julia> @m 2
An Integer

julia> x = 2
2

julia> @m x
1 arguments
```

18.4 Code Generation

When a significant amount of repetitive boilerplate code is required, it is common to generate it programmatically to avoid redundancy. In most languages, this requires an extra build step, and a separate program to generate the repetitive code. In Julia, expression interpolation and `eval` allow such code generation to take place in the normal course of program execution. For example, consider the following custom type

```
struct MyNumber
    x::Float64
end
# output
```

for which we want to add a number of methods to. We can do this programmatically in the following loop:

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(quote
        Base.$op(a::MyNumber) = MyNumber($op(a.x))
    end)
end
# output
```

and we can now use those functions with our custom type:

```
julia> x = MyNumber(π)
MyNumber(3.141592653589793)

julia> sin(x)
MyNumber(1.2246467991473532e-16)

julia> cos(x)
MyNumber(-1.0)
```

In this manner, Julia acts as its own preprocessor, and allows code generation from inside the language. The above code could be written slightly more tersely using the `:` prefix quoting form:

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(:(Base.$op(a::MyNumber) = MyNumber($op(a.x))))
end
```

This sort of in-language code generation, however, using the `eval(quote(...))` pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

```
for op = (:sin, :cos, :tan, :log, :exp)
    @eval Base.$op(a::MyNumber) = MyNumber($op(a.x))
end
```

The `@eval` macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to `@eval` can be a block:

```
@eval begin
    # multiple lines
end
```

18.5 Non-Standard String Literals

Recall from Strings that string literals prefixed by an identifier are called non-standard string literals, and can have different semantics than un-prefixed string literals. For example:

- `r"^\s*(?:#¦$)"` produces a regular expression object rather than a string
- `b"DATA\xff\u2200"` is a byte array literal for `[68,65,84,65,255,226,136,128]`.

Perhaps surprisingly, these behaviors are not hard-coded into the Julia parser or compiler. Instead, they are custom behaviors provided by a general mechanism that anyone can use: prefixed string literals are parsed as calls to specially-named macros. For example, the regular expression macro is just the following:

```
macro r_str(p)
    Regex(p)
end
```

That's all. This macro says that the literal contents of the string literal `r"^\s*(?:#¦$)"` should be passed to the `@r_str` macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression `r"^\s*(?:#¦$)"` is equivalent to placing the following object directly into the syntax tree:

```
Regex("^\\s*(?:#¦\$)")
```

Not only is the string literal form shorter and far more convenient, but it is also more efficient: since the regular expression is compiled and the `Regex` object is actually created when the code is compiled, the compilation occurs only once, rather than every time the code is executed. Consider if the regular expression occurs in a loop:

```
for line = lines
    m = match(r"^\s*(?:#¦$)", line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Since the regular expression `r"^\s*(?:#¦$)"` is compiled and inserted into the syntax tree when this code is parsed, the expression is only compiled once instead of each time the loop is executed. In order to accomplish this without macros, one would have to write this loop like this:

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
    m = match(re, line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Moreover, if the compiler could not determine that the regex object was constant over all loops, certain optimizations might not be possible, making this version still less efficient than the more convenient literal form above. Of course, there are still situations where the non-literal form is more convenient: if one needs to interpolate a variable into the regular expression, one must take this more verbose approach; in cases where the regular expression pattern itself is dynamic, potentially changing upon each loop iteration, a new regular expression object must be constructed on each iteration. In the vast majority of use cases, however, regular expressions are not constructed based on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is invaluable.

Like non-standard string literals, non-standard command literals exist using a prefixed variant of the command literal syntax. The command literal `custom`literal`` is parsed as `@custom_cmd "literal"`. Julia itself does not contain any non-standard command literals, but packages can make use of this syntax. Aside from the different syntax and the `_cmd` suffix instead of the `_str` suffix, non-standard command literals behave exactly like non-standard string literals.

In the event that two modules provide non-standard string or command literals with the same name, it is possible to qualify the string or command literal with a module name. For instance, if both `Foo` and `Bar` provide non-standard string literal `@x_str`, then one can write `Foo.x"literal"` or `Bar.x"literal"` to disambiguate between the two.

The mechanism for user-defined string literals is deeply, profoundly powerful. Not only are Julia's non-standard literals implemented using it, but also the command literal syntax (`` `echo "Hello, $person"` ``) is implemented with the following innocuous-looking macro:

```
macro cmd(str)
    :(cmd_gen($(shell_parse(str)[1])))
end
```

Of course, a large amount of complexity is hidden in the functions used in this macro definition, but they are just functions, written entirely in Julia. You can read their source and see precisely what they do – and all they do is construct expression objects to be inserted into your program's syntax tree.

18.6 Generated functions

A very special macro is `@generated`, which allows you to define so-called generated functions. These have the capability to generate specialized code depending on the types of their arguments with more flexibility and/or less code than what can be achieved with multiple dispatch. While macros work with expressions at parse time and cannot access the types of their inputs, a generated function gets expanded at a time when the types of the arguments are known, but the function is not yet compiled.

Instead of performing some calculation or action, a generated function declaration returns a quoted expression which then forms the body for the method corresponding to the types of the arguments. When a generated function is called, the expression it returns is compiled and then run. To make this efficient, the result is usually cached. And to make this inferable, only a limited subset of the language is usable. Thus, generated functions provide a flexible way to move work from run time to compile time, at the expense of greater restrictions on allowed constructs.

When defining generated functions, there are five main differences to ordinary functions:

1. You annotate the function declaration with the `@generated` macro. This adds some information to the AST that lets the compiler know that this is a generated function.
2. In the body of the generated function you only have access to the types of the arguments — not their values.
3. Instead of calculating something or performing some action, you return a quoted expression which, when evaluated, does what you want.
4. Generated functions are only permitted to call functions that were defined before the definition of the generated function. (Failure to follow this my result on getting `MethodErrors` referring to functions from a future world-age.)
5. Generated functions must not mutate or observe any non-constant global state (including, for example, IO, locks, non-local dictionaries, or using `hasmethod`). This means they can only read global constants, and cannot have any side effects. In other words, they must be completely pure. Due to an implementation limitation, this also means that they currently cannot define a closure or generator.

It's easiest to illustrate this with an example. We can declare a generated function `foo` as

```
julia> @generated function foo(x)
           Core.println(x)
           return :(x * x)
       end
foo (generic function with 1 method)
```

Note that the body returns a quoted expression, namely `:(x * x)`, rather than just the value of `x * x`.

From the caller's perspective, this is identical to a regular function; in fact, you don't have to know whether you're calling a regular or generated function. Let's see how `foo` behaves:

```
julia> x = foo(2); # note: output is from println() statement in the body
Int64

julia> x           # now we print x
4

julia> y = foo("bar");
String

julia> y
"barbar"
```

So, we see that in the body of the generated function, `x` is the type of the passed argument, and the value returned by the generated function, is the result of evaluating the quoted expression we returned from the definition, now with the value of `x`.

What happens if we evaluate `foo` again with a type that we have already used?

```
julia> foo(4)
16
```

Note that there is no printout of `Int64`. We can see that the body of the generated function was only executed once here, for the specific set of argument types, and the result was cached. After that, for this example, the expression returned from the generated function on the first invocation was re-used as the method body. However, the actual caching behavior is an implementation-defined performance optimization, so it is invalid to depend too closely on this behavior.

The number of times a generated function is generated might be only once, but it might also be more often, or appear to not happen at all. As a consequence, you should never write a generated function with side effects - when, and how often, the side effects occur is undefined. (This is true for macros too - and just like for macros, the use of `eval` in a generated function is a sign that you're doing something the wrong way.) However, unlike macros, the runtime system cannot correctly handle a call to `eval`, so it is disallowed.

It is also important to see how `@generated` functions interact with method redefinition. Following the principle that a correct `@generated` function must not observe any mutable state or cause any mutation of global state, we see the

following behavior. Observe that the generated function cannot call any method that was not defined prior to the definition of the generated function itself.

Initially `f(x)` has one definition

```
julia> f(x) = "original definition";
```

Define other operations that use `f(x)`:

```
julia> g(x) = f(x);

julia> @generated gen1(x) = f(x);

julia> @generated gen2(x) = :(f(x));
```

We now add some new definitions for `f(x)`:

```
julia> f(x::Int) = "definition for Int";

julia> f(x::Type{Int}) = "definition for Type{Int}";
```

and compare how these results differ:

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> gen1(1)
"original definition"

julia> gen2(1)
"definition for Int"
```

Each method of a generated function has its own view of defined functions:

```
julia> @generated gen1(x::Real) = f(x);

julia> gen1(1)
"definition for Type{Int}"
```

The example generated function `foo` above did not do anything a normal function `foo(x) = x * x` could not do (except printing the type on the first invocation, and incurring higher overhead). However, the power of a generated function lies in its ability to compute different quoted expressions depending on the types passed to it:

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

(although of course this contrived example would be more easily implemented using multiple dispatch...)

Abusing this will corrupt the runtime system and cause undefined behavior:

```
julia> @generated function baz(x)
           if rand() < .9
               return :(x^2)
           else
               return :("boo!")
           end
       end
baz (generic function with 1 method)
```

Since the body of the generated function is non-deterministic, its behavior, and the behavior of all subsequent code is undefined.

Don't copy these examples!

These examples are hopefully helpful to illustrate how generated functions work, both in the definition end and at the call site; however, don't copy them, for the following reasons:

- the `foo` function has side-effects (the call to `Core.println`), and it is undefined exactly when, how often or how many times these side-effects will occur
- the `bar` function solves a problem that is better solved with multiple dispatch - defining `bar(x) = x` and `bar(x::Integer) = x ^ 2` will do the same thing, but it is both simpler and faster.
- the `baz` function is pathological

Note that the set of operations that should not be attempted in a generated function is unbounded, and the runtime system can currently only detect a subset of the invalid operations. There are many other operations that will simply corrupt the runtime system without notification, usually in subtle ways not obviously connected to the bad definition. Because the function generator is run during inference, it must respect all of the limitations of that code.

Some operations that should not be attempted include:

1. Caching of native pointers.
2. Interacting with the contents or methods of `Core.Compiler` in any way.
3. Observing any mutable state.
    - Inference on the generated function may be run at any time, including while your code is attempting to observe or mutate this state.
4. Taking any locks: C code you call out to may use locks internally, (for example, it is not problematic to call `malloc`, even though most implementations require locks internally) but don't attempt to hold or acquire any while executing Julia code.
5. Calling any function that is defined after the body of the generated function. This condition is relaxed for incrementally-loaded precompiled modules to allow calling any function in the module.

Alright, now that we have a better understanding of how generated functions work, let's use them to build some more advanced (and valid) functionality...

An advanced example

Julia's base library has a an internal `sub2ind` function to calculate a linear index into an n-dimensional array, based on a set of n multilinear indices - in other words, to calculate the index `i` that can be used to index into an array `A` using `A[i]`, instead of `A[x,y,z,...]`. One possible implementation is the following:

```
julia> function sub2ind_loop(dims::NTuple{N}, I::Integer...) where N
           ind = I[N] - 1
           for i = N-1:-1:1
               ind = I[i]-1 + dims[i]*ind
           end
           return ind + 1
       end
sub2ind_loop (generic function with 1 method)

julia> sub2ind_loop((3, 5), 1, 2)
4
```

The same thing can be done using recursion:

```
julia> sub2ind_rec(dims::Tuple{}) = 1;

julia> sub2ind_rec(dims::Tuple{}, i1::Integer, I::Integer...) =
           i1 == 1 ? sub2ind_rec(dims, I...) : throw(BoundsError());

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer) = i1;

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer, I::Integer...) =
           i1 + dims[1] * (sub2ind_rec(Base.tail(dims), I...) - 1);

julia> sub2ind_rec((3, 5), 1, 2)
4
```

Both these implementations, although different, do essentially the same thing: a runtime loop over the dimensions of the array, collecting the offset in each dimension into the final index.

However, all the information we need for the loop is embedded in the type information of the arguments. Thus, we can utilize generated functions to move the iteration to compile-time; in compiler parlance, we use generated functions to manually unroll the loop. The body becomes almost identical, but instead of calculating the linear index, we build up an expression that calculates the index:

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen (generic function with 1 method)

julia> sub2ind_gen((3, 5), 1, 2)
4
```

What code will this generate?

An easy way to find out is to extract the body into another (regular) function:

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           return sub2ind_gen_impl(dims, I...)
       end
sub2ind_gen (generic function with 1 method)

julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <: NTuple{N,Any} where N
           length(I) == N || return :(error("partial indexing is unsupported"))
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen_impl (generic function with 1 method)
```

We can now execute `sub2ind_gen_impl` and examine the expression it returns:

```
julia> sub2ind_gen_impl(Tuple{Int,Int}, Int, Int)
:(((I[1] - 1) + dims[1] * (I[2] - 1)) + 1)
```

So, the method body that will be used here doesn't include a loop at all - just indexing into the two tuples, multiplication and addition/subtraction. All the looping is performed compile-time, and we avoid looping during execution entirely. Thus, we only loop once per type, in this case once per `N` (except in edge cases where the function is generated more than once - see disclaimer above).

Optionally-generated functions

Generated functions can achieve high efficiency at run time, but come with a compile time cost: a new function body must be generated for every combination of concrete argument types. Typically, Julia is able to compile "generic" versions of functions that will work for any arguments, but with generated functions this is impossible. This means that programs making heavy use of generated functions might be impossible to statically compile.

To solve this problem, the language provides syntax for writing normal, non-generated alternative implementations of generated functions. Applied to the `sub2ind` example above, it would look like this:

```
function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
    if N != length(I)
        throw(ArgumentError("Number of dimensions must match number of indices."))
    end
    if @generated
        ex = :(I[$N] - 1)
        for i = (N - 1):-1:1
            ex = :(I[$i] - 1 + dims[$i] * $ex)
        end
        return :($ex + 1)
    else
        ind = I[N] - 1
        for i = (N - 1):-1:1
            ind = I[i] - 1 + dims[i]*ind
        end
        return ind + 1
    end
end
```

Internally, this code creates two implementations of the function: a generated one where the first block in `if @generated` is used, and a normal one where the `else` block is used. Inside the `then` part of the `if @generated` block, code has the same semantics as other generated functions: argument names refer to types, and the code should return an expression. Multiple `if @generated` blocks may occur, in which case the generated implementation uses all of the `then` blocks and the alternate implementation uses all of the `else` blocks.

Notice that we added an error check to the top of the function. This code will be common to both versions, and is run-time code in both versions (it will be quoted and returned as an expression from the generated version). That means that the values and types of local variables are not available at code generation time -- the code-generation code can only see the types of arguments.

In this style of definition, the code generation feature is essentially an optional optimization. The compiler will use it if convenient, but otherwise may choose to use the normal implementation instead. This style is preferred, since it allows the compiler to make more decisions and compile programs in more ways, and since normal code is more readable than code-generating code. However, which implementation is used depends on compiler implementation details, so it is essential for the two implementations to behave identically.

Chapter 19

다차원 배열

Julia는 대부분의 수치 계산용 언어처럼 일급 객체로써 배열 구현을 제공한다. 대부분의 언어는 배열의 구현에 많은 신경을 쓰지만 Julia는 배열을 특별하게 취급하지는 않는다. 배열 관련 라이브러리는 거의 다 Julia로 작성되었으며, Julia 컴파일러가 성능을 좌우한다. 또한 `AbstractArray`를 상속하여 커스텀 배열 타입을 정의하는 것이 가능하다. 커스텀 배열 타입을 구현할 때 필요한 세부사항은 manual section on the AbstractArray interface 를 참조하기 바란다.

배열은 다차원 그리드(multi-dimensional grid)에 저장된 객체들의 모음이다. 일반으로 배열은 `Any` 타입의 객체들을 담을 수 있다. 수치 계산용으로 사용하려면 배열은 `Float64`이나 `Int32`와 같이 더 구체적인 타입의 객체를 담는 것이 좋다.

다른 많은 수치 계산용 언어에서는 성능을 위해 프로그램을 벡터화된 스타일로 작성하지만, Julia에서는 대개 그럴 필요가 없다. Julia 컴파일러는 타입 추론을 통해 스칼라 배열 인덱싱에 최적화된 코드를 생성한다. 따라서 편리하고 읽기 쉬운 스타일로 프로그램을 작성해도 성능이 보존되며, 오히려 메모리를 더 적게 사용하는 경우도 있다.

줄리아는 함수에 인자를 줄 때 "공유를 통한 전달(pass-by-sharing)을 한다. 몇몇 프로그래밍 언어는 배열을 값으로 전달하여(pass-by-value), 원치않는 수정을 막지만 무분별한 값의 복사로 인해 속도 지연을 겪을 수 있다. Julia는 함수 내에서 값이 수정되거나 삭제될 가능성을 있을 때, 관습적으로 `!`을 함수 이름의 마지막에 붙여서 알려준다(`sort`와 `sort!`를 비교해보자). Collee가 객체의 수정을 피하려면 명시적으로 객체를 복사를 해야한다. 객체를 수정하지 않는 함수는 `!`이 붙여진 동일 이름의 함수와 같은 역할을 하면서 복사된 객체를 반환한다.

19.1 기본 함수

19.2 생성과 초기화

배열을 생성하고 초기화 하는 많은 함수가 있다. 다음에 나열된 함수들에서 `dims...` 인수는 차원의 크기들을 나타내는 튜플 하나를 받거나, 혹은 각 차원의 크기를 여러 인수로 받을 수 있다. 이 함수들의 대부분은 첫번째 인수로 배열의 원소 타입 `T`를 받을 수 있다. `T`가 생략되었다면 `Float64`가 기본값이다.

`[A, B, C, ...]` 문법은 주어진 인수들의 일차원 배열(이를테면 벡터)을 생성한다. 만약 모든 인수가 공통의 확장 타입(promotion type)을 가진다면, 이들은 `convert`를 통해 공통의 확장 타입으로 변환된다.

함수	설명
eltype(A)	A 의 원소 타입
length(A)	A 의 원소 갯수
ndims(A)	A 의 차원수
size(A)	A 의 크기 튜플
size(A,n)	A 의 n 차원의 크기
axes(A)	A 의 유효한 인덱스 튜플
axes(A,n)	A 의 유효 인덱스 n차원 범위(range)
eachindex(A)	A 의 모든 위치를 방문하는 효율적인 반복자(iterator)
stride(A,k)	k 차원 방향의 스트라이드 (연속한 원소 간의 선형 인덱스 거리)
strides(A)	모든 차원의 스트라이드 튜플

To see the various ways we can pass dimensions to these constructors, consider the following examples:

```
julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros(Int8, (2, 3))
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros((2, 3))
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
```

Here, `(2, 3)` is a `Tuple`.

19.3 병합(Concatenation)

배열은 다음의 함수를 사용하여 생성하고 병합할 수 있다.

스칼라 값이 인수로 전달되면 원소 갯수가 하나인 배열로 취급한다. 예를 들어,

함수	설명
`Array{T}(undef, dims...)`	초기화 되지 않은 밀집 `Array`
`zeros(T, dims...)`	모든 값이 0으로 초기화 된 `Array`
`ones(T, dims...)`	모든 값이 1로 초기화 된 `Array`
`trues(dims...)`	모든 값이 `true`로 초기화 된 `BitArray`
`falses(dims...)`	모든 값이 `false`로 초기화 된 `BitArray`
`reshape(A, dims...)`	`A` 와 동일한 데이타를 가지고 있지만 형상이 다른 배열
`copy(A)`	`A` 의 얕은 복사
`deepcopy(A)`	`A` 의 깊은 복사 (모든 원소를 재귀적으로 복사함)
`similar(A, T, dims...)`	`A` 와 동일한 종류(밀집, 희소, 등)의 초기화 되지 않은 배열. 지정된 원소 타입과 형상을 가짐. 두번째와 세번째 인수는 선택적이며, 기본값은 `A`의 원소타입과 차원이다.
`reinterpret(T, A)`	`A` 와 동일한 이진 데이터를 가지고 있지만, 원소 타입이 `T` 인 배열
`rand(T, dims...)`	독립 동일하며 열린구간 $[0, 1)$ 상에서 연속 균일 분포를 가진 랜덤 `Array`
`randn(T, dims...)`	독립 동일하며 표준 정규 분포를 가진 랜덤 `Array`
`Matrix{T}(I, m, n)`	크기가 `m` × `n` 인 단위 행렬
`range(start, stop=stop, length=n)`	`start`에서 `stop`까지 `n` 개의 원소가 선형적으로 배치된 구간
`fill!(A, x)`	배열 `A` 를 `x` 값으로 채우기
`fill(x, dims...)`	`x` 값으로 차 있는 `Array`

함수	설명
`cat(A...; dims=k)`	입력 배열을 `k` 차원에 따라 병합
`vcat(A...)`	`cat(A...; dims=1)`의 줄임
`hcat(A...)`	`cat(A...; dims=2)`의 줄임

```
julia> vcat([1, 2], 3)
3-element Array{Int64,1}:
 1
 2
 3

julia> hcat([1 2], 3)
1×3 Array{Int64,2}:
```

```
1  2  3
```

병합 함수는 자주 사용되므로 다음의 특별한 문법을 가진다:

표현식	함수
`[A; B; C; ...]`	vcat
`[A B C ...]`	hcat
`[A B; C D; ...]`	hvcat

hvcat 은 1차원 (세미콜론으로 구분) 과 2차원(스페이스로 구분) 모두 병합한다. 아래 예제의 문법과 결과물을 비교해보자:

```
julia> [[1; 2]; [3, 4]]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> [[1 2] [3 4]]
1×4 Array{Int64,2}:
 1  2  3  4

julia> [[1 2]; [3 4]]
2×2 Array{Int64,2}:
 1  2
 3  4
```

19.4 타입이 있는 배열의 초기화

특정 원소 타입의 배열은 `T[A, B, C, ...]` 문법을 통해 생성할 수 있다. 이는 원소 타입이 `T`인 일차원 배열을 생성하고, 원소 `A`, `B`, `C` 등을 담도록 초기화한다. 예를 들어, `Any[x, y, z]`는 어떤 값이든 가질 수 있는 배열을 생성한다.

병합 구문 또한 비슷한 방법으로 원소 타입을 지정할 수 있다.

```
julia> [[1 2] [3 4]]
1×4 Array{Int64,2}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
```

```
1×4 Array{Int8,2}:
 1  2  3  4
```

19.5 컴프리헨션(Comprehensions)

컴프리헨션은 배열을 생성하는 일반적이면서도 강력한 방법을 제공한다. 컴프리헨션의 문법은 수학에서 쓰이는 집합의 조건제시법과 유사하다:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

이는 x, y 등의 변수가 주어진 목록의 값을 각각 가지도록 하여 F(x,y,...)를 계산한다는 뜻이다. 값의 목록은 반복 가능한(iterable) 어떤 객체도 될 수 있지만, 주로 1:n 혹은 2:(n-1) 와 같은 범위이거나, [1.2, 3.4, 5.7]와 같이 명시적인 값의 배열이다. 결과는 N차원 배열이며, 그 크기는 변수의 범위인 rx, ry 등의 크기를 병합한 것과 같다. 그리고 각 F(x,y,...) 계산은 스칼라 값을 리턴한다.

다음의 예는 일차원 그리드에서, 현재 원소와 왼쪽 이웃, 오른쪽 이웃의 가중 평균을 계산한다:

```
julia> x = rand(8)
8-element Array{Float64,1}:
 0.843025
 0.869052
 0.365105
 0.699456
 0.977653
 0.994953
 0.41084
 0.809411

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Array{Float64,1}:
 0.736559
 0.57468
 0.685417
 0.912429
 0.8446
 0.656511
```

결과 배열의 타입은 계산된 원소가 결정한다. 타입을 명시적으로 정하려면 컴프리헨션 앞에 타입을 붙이면 된다. 예를 들어, 다음과 같이 결과를 단정밀도(single precision)로 요청할 수 있다:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

19.6 제너레이터 표현식 (Generator Expressions)

컴프리헨션은 대괄호 없이도 쓸 수 있으며, 이 경우 제너레이터 객체를 생성한다. 제너레이터는 배열을 미리 할당하고 값을 저장하는 것이 아니라, 필요에 따라 값을 생성하도록 반복할 수 있다 (반복 참조). 예를 들어, 다음의 표현식은 메모리 할당 없이 수열의 합을 계산한다:

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

인수 목록 안에서 다차원 제너레이터 표현식을 사용할 때에는 괄호를 사용하여 그 다음의 인수와 구분한다:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

`for` 다음에 나오는 쉼표로 구분된 모든 표현식은 범위로 해석되므로, 여기에 괄호를 추가함으로써 `map`에 세번째 인수를 추가할 수 있다.

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Array{Tuple{Float64,Int64},2}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

Generators are implemented via inner functions. Just like inner functions used elsewhere in the language, variables from the enclosing scope can be "captured" in the inner function. For example, `sum(p[i] - q[i] for i=1:n)` captures the three variables `p`, `q` and `n` from the enclosing scope. Captured variables can present performance challenges; see performance tips.

제너레이터와 컴프리헨션에서 `for` 키워드를 여러번 사용함으로써 범위가 앞선 범위에 의존하도록 할 수 있다.

```
julia> [(i,j) for i=1:3 for j=1:i]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

이러한 경우 결과는 항상 1차원이다.

생성된 값은 `if` 키워드를 사용하여 필터링 할 수 있다.

```
julia> [(i,j) for i=1:3 for j=1:i if i+j == 4]
2-element Array{Tuple{Int64,Int64},1}:
 (2, 2)
 (3, 1)
```

19.7 인덱싱

n차원 배열 `A`를 인덱싱 하는 일반적인 문법은 다음과 같다:

```
X = A[I_1, I_2, ..., I_n]
```

여기서 `I_k` 는 스칼라 정수, 정수의 배열, 혹은 지원하는 다른 인덱스 중 하나이다. 여기에는 모든 인덱스를 선택하는 `Colon` (`:`), 연속되거나 일정한 간격의 부분수열을 선택하는 `a:c` 혹은 `a:b:c`와 같은 형태의 범위, 그리고 `true` 값을 선택하는 부울 배열도 포함된다.

만약 모든 인덱스가 스칼라라면, 결과 `X`는 배열 `A`의 원소 중 하나이다. 그렇지 않을 경우 `X`는 배열이며, 모든 인덱스의 차원 수의 합이 `X`의 차원수가 된다.

예를 들어, 모든 인덱스 `I_k`가 벡터라면 `X`의 크기는 `(length(I_1), length(I_2), ..., length(I_n))`가 되고, `X`의 `i_1, i_2, ..., i_n` 위치는 `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]` 값을 가지게 된다.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12
```

```
[:, :, 2, 2] =
 13  15
 14  16

julia> A[1, 2, 1, 1] # all scalar indices
3

julia> A[[1, 2], [1], [1, 2], [1]] # all vector indices
2×1×2×1 Array{Int64,4}:
[:, :, 1, 1] =
 1
 2

[:, :, 2, 1] =
 5
 6

julia> A[[1, 2], [1], [1, 2], 1] # a mix of index types
2×1×2 Array{Int64,3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 5
 6
```

Note how the size of the resulting array is different in the last two cases.

만약 `I_1`이 2차원 행렬로 바뀐다면, `X`는 크기가 `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`인 `n+1`차원 배열이 된다. 행렬이 차원을 하나 추가하는 것이다.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2));

julia> A[[1 2; 1 2]]
2×2 Array{Int64,2}:
 1  2
 1  2
```

```
julia> A[[1 2; 1 2], 1, 2, 1]
2×2 Array{Int64,2}:
 5  6
 5  6
```

X의 `i_1, i_2, i_3, ..., i_{n+1}` 위치는 `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]` 값을 가진다. 스칼라(scalars)로 인덱스된 모든 차원은 결과에서 빠진다. 예를 들어, `J`가 인덱스들의 배열이면 `A[2, J, 3]`의 결과는 크기가 `size(J)`인 배열이며, `j`번째 원소의 값은 `A[2, J[j], 3]`이다.

인덱싱 문법의 특수한 한 부분으로서, 각 차원의 마지막 인덱스를 나타내기 위해서 인덱싱 괄호 안에서 `end` 키워드를 사용할 수 있다. 마지막 인덱스는 인덱싱 되는 가장 안쪽의 배열의 크기에 따라 결정된다. `end` 키워드 없는 인덱싱 문법은 `getindex` 호출과 동일하다:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

예시:

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2×2 Array{Int64,2}:
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Array{Int64,2}:
  5  9
 13  1
```

19.8 대입

n차원 배열 `A`에 값을 대입하는 일반적인 문법은 다음과 같다:

```
A[I_1, I_2, ..., I_n] = X
```

여기서 `I_k` 는 스칼라 정수, 정수의 배열, 혹은 지원하는 다른 인덱스 중 하나이다. 여기에는 모든 인덱스를 선택하는 `Colon` (`:`), 연속되거나 일정한 간격의 부분수열을 선택하는 `a:c` 혹은 `a:b:c`와 같은 형태의 범위, 그리고 `true` 값을 선택하는 부울 배열도 포함된다.

만약 인덱스 `I_k`들이 모두 정수라면, `A`의 `I_1, I_2, ..., I_n`에 위치한 값은 `X`의 값으로 덮어씌워진다. 필요하다면 `A`의 `eltype`으로 형 변환`convert`이 일어날 수 있다.

만약 어떤 인덱스 `I_k`가 한 개보다 많은 위치를 선택할 경우, 우변의 `X`는 `A[I_1, I_2, ..., I_n]`의 인덱싱 결과와 같은 모양의 배열이거나, 성분 개수가 같은 벡터여야만 한다. `A`의 `I_1[i_1], I_2[i_2], ..., I_n[i_n]`에 위치한 값은 값 `X[I_1, I_2, ..., I_n]`으로 덮어씌워지며, 필요한 경우 형 변환이 일어난다. 성분별 대입 연산자(The element-wise assignment operator) `.=`를 선택된 위치 전체에 대한 `X`의 브로드캐스팅broadcast에 쓸 수도 있다.

```
A[I_1, I_2, ..., I_n] .= X
```

인덱싱에서와 마찬가지로, 각 차원의 마지막 인덱스를 나타내기 위해서 인덱싱 괄호 안에서 `end` 키워드를 사용할 수 있다. 마지막 인덱스는 인덱싱 되는 가장 안쪽의 배열의 크기에 따라 결정된다. `end` 키워드 없는 대입의 문법은 `setindex!` 호출과 동일하다:

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

예시:

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[3, 3] = -9;

julia> x[1:2, 1:2] = [-1 -4; -2 -5];

julia> x
3×3 Array{Int64,2}:
 -1  -4   7
 -2  -5   8
  3   6  -9
```

19.9 지원하는 인덱스 타입

표현식 `A[I_1, I_2, ..., I_n]`에서, `I_k`는 스칼라 인덱스, 스칼라 인덱스의 배열, 혹은 `to_indices`를 통해 스칼라 인덱스 배열로 변환될 수 있는 객체 중 하나이다:

1. 스칼라 인덱스. 다음을 포함한다:
   - 부울이 아닌 정수.
   - `CartesianIndex{N}`. 여러 차원에 걸쳐있는 정수의 `N`튜플처럼 행동한다. (자세한 내용은 아래를 참조.)
2. 스칼라 인덱스의 배열. 다음을 포함한다:
   - 정수 벡터와 다차원 정수 배열.
   - `[]`와 같은 빈 배열. 아무 원소도 선택하지 않는다.
   - `a:c` 나 `a:b:c`와 같은 범위. `a` 와 `c` 사이의 연속되거나 일정 간격의 subsection을 선택.
   - `AbstractArray`의 서브타입인 배열 스칼라의 .
   - `CartesianIndex{N}`의 배열 (자세한 내용은 아래를 참조).
3. 스칼라 인덱스의 배열을 나타내는 객체이면서 `to_indices`를 통해 스칼라 인덱스의 배열로 변환될 수 있는 것. 기본으로 다음을 포함한다:
   - `Colon()` (`:`). 차원 혹은 배열의 모든 원소를 선택한다.
   - 부울 배열. `true` 인덱스에 있는 원소를 선택한다 (자세한 내용은 아래를 참조)

Some examples:

```
julia> A = reshape(collect(1:2:18), (3, 3))
3×3 Array{Int64,2}:
 1   7  13
 3   9  15
 5  11  17

julia> A[4]
7

julia> A[[2, 5, 8]]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[[1 4; 3 8]]
2×2 Array{Int64,2}:
 1   7
```

```
 5  15

julia> A[[]]
0-element Array{Int64,1}

julia> A[1:2:5]
3-element Array{Int64,1}:
 1
 5
 9

julia> A[2, :]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[:, 3]
3-element Array{Int64,1}:
 13
 15
 17
```

직교 인덱스(Cartesian indices)

`CartesianIndex{N}` 객체는 여러 차원을 포괄하는 정수의 `N`튜플처럼 동작하는 스칼라 인덱스를 나타낸다.

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

따로 떼어놓고 생각했을때, 이는 매우 간단하게 보일지도 모른다; `CartesianIndex`는 단순히 여러 정수를 하나의 객체로 묶어서 하나의 다차원 인덱스로 나타내는 것이다. 하지만 다른 형식의 인덱싱이나, `CartesianIndex`를 내어놓는 반복자와 결합하면 매우 우아하고 효율적인 코드를 쓸 수 있다. 아래의 반복자를 참조하라. 더 고급 예시는 다차원 알고리즘과 반복에 관한 이 블로그를 참조하라.

`CartesianIndex{N}`의 배열 또한 지원된다. 이는 각각 `N`차원을 포괄하는 스칼라 인덱스의 모음을 나타나며, 점별(pointwise) 인덱싱이라고도 불리는 인덱싱의 형태를 가능하게 한다. 예를 들어, 앞선 예시에서 정의된 `A`의 첫 "페이지"의 대각원소들을 다음과 같이 엑세스 할 수 있다:

```
julia> page = A[:,:,1]
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> page[[CartesianIndex(1,1),
             CartesianIndex(2,2),
             CartesianIndex(3,3),
             CartesianIndex(4,4)]]
4-element Array{Int64,1}:
  1
  6
 11
 16
```

이는 점 브로드캐스팅을 정수 인덱스와 함께 씀으로써 (`A`로 부터 첫번째 "페이지"를 추출하는 별도의 과정 없이) 더욱더 간단하게 표한할 수 있다. 뿐만 아니라 `:`와 결합하여 두 페이지의 대각원소들을 한번에 추출할 수도 있다:

```
julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), 1]
4-element Array{Int64,1}:
  1
  6
 11
 16

julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), :]
4×2 Array{Int64,2}:
  1  17
  6  22
 11  27
 16  32
```

!!! 경고

> `CartesianIndex와` `CartesianIndex의` 배열은 차원의 마지막 인덱스를 나타내는 `end` 키워드와 호환되지 않으므로, `CartesianIndex` 혹은 `CartesianIndex의` 배열을 포함할 수도 있는 표현식에서는 `end를` 사용해서는 안된다.

논리적 인덱싱

부울 배열을 이용한 인덱싱은 값이 `true`인 곳의 인덱스를 선택한다. 주로 논리적 인덱싱, 혹은 논리적 마스크를 사용한 인덱싱이라고 부르며, 부울 벡터 `B`를 통한 인덱싱은 `findall(B)`가 리턴하는 정수의 벡터를 통한 인덱싱과 동일하다. 이와 마찬가지로, `N`차원 부울 배열을 통한 인덱싱은, `true` 값의 위치를 나타내는 `CartesianIndex{N}`들의 배열을 통한 인덱싱과 동일하다. 논리적 인덱스는, 인덱스의 크기와 인덱스하는 배열의 해당 차원의 크기가 일치하거나, 혹은 배열과 크기 및 차원이 일치하는 단 하나의 인덱스이어야 한다. 부울 배열을 사용하여 바로 인덱싱 하는 것이 `findall`를 먼저 호출하는 것보다 일반적으로 더 효율적이다.

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[[false, true, true, false], :]
2×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Array{Bool,2}:
 1  0  0  0
 1  0  0  0
 0  0  0  0
 1  1  0  1

julia> x[mask]
5-element Array{Int64,1}:
  1
  2
  4
  8
 16
```

Number of indices

Cartesian indexing

The ordinary way to index into an `N`-dimensional array is to use exactly `N` indices; each index selects the position(s) in its particular dimension. For example, in the three-dimensional array `A = rand(4, 3, 2)`, `A[2, 3, 1]` will select the number in the second row of the third column in the first "page" of the array. This is often referred to as cartesian indexing.

Linear indexing

When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array. Instead, it selects the `i`th element using the column-major iteration order that linearly spans the entire array. This is known as linear indexing. It essentially treats the array as though it had been reshaped into a one-dimensional vector with `vec`.

```
julia> A = [2 6; 4 7; 3 1]
3×2 Array{Int64,2}:
 2  6
 4  7
 3  1

julia> A[5]
7

julia> vec(A)[5]
7
```

A linear index into the array `A` can be converted to a `CartesianIndex` for cartesian indexing with `CartesianIndices(A)[i]` (see `CartesianIndices`), and a set of `N` cartesian indices can be converted to a linear index with `LinearIndices(A)[i_1, i_2, ..., i_N]` (see `LinearIndices`).

```
julia> CartesianIndices(A)[5]
CartesianIndex(2, 2)

julia> LinearIndices(A)[2, 2]
5
```

It's important to note that there's a very large assymmetry in the performance of these conversions. Converting a linear index to a set of cartesian indices requires dividing and taking the remainder, whereas going the other way is

just multiplies and adds. In modern processors, integer division can be 10-50 times slower than multiplication. While some arrays — like `Array` itself — are implemented using a linear chunk of memory and directly use a linear index in their implementations, other arrays — like `Diagonal` — need the full set of cartesian indices to do their lookup (see `IndexStyle` to introspect which is which). As such, when iterating over an entire array, it's much better to iterate over `eachindex(A)` instead of `1:length(A)`. Not only will the former be much faster in cases where `A` is `IndexCartesian`, but it will also support OffsetArrays, too.

Omitted and extra indices

In addition to linear indexing, an `N`-dimensional array may be indexed with fewer or more than `N` indices in certain situations.

Indices may be omitted if the trailing dimensions that are not indexed into are all length one. In other words, trailing indices can be omitted only if there is only one possible value that those omitted indices could be for an in-bounds indexing expression. For example, a four-dimensional array with size `(3, 4, 2, 1)` may be indexed with only three indices as the dimension that gets skipped (the fourth dimension) has length one. Note that linear indexing takes precedence over this rule.

```
julia> A = reshape(1:24, 3, 4, 2, 1)
3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype Int64:
[:, :, 1, 1] =
 1  4  7  10
 2  5  8  11
 3  6  9  12

[:, :, 2, 1] =
 13  16  19  22
 14  17  20  23
 15  18  21  24

julia> A[1, 3, 2] # Omits the fourth dimension (length 1)
19

julia> A[1, 3] # Attempts to omit dimensions 3 & 4 (lengths 2 and 1)
ERROR: BoundsError: attempt to access 3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype Int64 at index [1,
↪  3]

julia> A[19] # Linear indexing
19
```

When omitting all indices with `A[]`, this semantic provides a simple idiom to retrieve the only element in an array and simultaneously ensure that there was only one element.

Similarly, more than `N` indices may be provided if all the indices beyond the dimensionality of the array are `1` (or more generally are the first and only element of `axes(A, d)` where `d` is that particular dimension number). This allows vectors to be indexed like one-column matrices, for example:

```
julia> A = [8,6,7]
3-element Array{Int64,1}:
 8
 6
 7

julia> A[2,1]
6
```

19.10 반복(Iteration)

배열 전체를 반복하는 방법으로는 다음을 추천한다:

```
for a in A
    # 원소 a로 뭔가 한다
end

for i in eachindex(A)
    # i 혹은 A[i] 로 뭔가 한다
end
```

첫번째 구문은 인덱스가 아니라 값이 필요할 때 사용한다. 두번째 구문에서 `A`가 빠른 선형 인덱싱을 지원하는 배열이라면 `i`는 `Int` 타입, 그렇지 않을 경우는 `CartesianIndex` 타입이다:

```
julia> A = rand(4,3);

julia> B = view(A, 1:3, 2:3);

julia> for i in eachindex(B)
           @show i
       end
i = CartesianIndex(1, 1)
```

```
i = CartesianIndex(2, 1)
i = CartesianIndex(3, 1)
i = CartesianIndex(1, 2)
i = CartesianIndex(2, 2)
i = CartesianIndex(3, 2)
```

`for i = 1:length(A)`에 비해, `eachindex`는 모든 종류의 배열을 효율적으로 반복할 수 있도록 해준다.

19.11 배열 특성(trait)

커스텀 `AbstractArray` 타입을 정의하는 경우, 빠른 선형 인덱싱이 가능함을 지정할 수 있다:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

이 설정은 `eachindex`가 정수를 사용하여 `MyArray`를 반복하도록 한다. 이 특성을 지정하지 않으면, 기본값인 `IndexCartesian()`를 사용한다.

19.12 배열과 벡터화된 연산자/함수

배열은 다음의 연산자를 지원한다:

1. 단항 산술 연산자 – `-`, `+`
2. 이항 산술 연산자 – `-`, `+`, `*`, `/`, `\`, `^`
3. 비교 연산자 – `==`, `!=`, ≈ (`isapprox`), ≠

배열 혹은 배열과 스칼라의 혼합에 대한 원소별 연산에 대해 `f.(args...)` 형태의 (예: `sin.(x)`, `min.(x,y)`) 점 문법을 사용하여 수학 연산과 다른 연산을 편리하게 벡터화 할 수 있다; 배열, 혹은 배열과 스칼라의 혼합에 대해 원소별 연산을 하기 위해서 점 문법을 쓸 수 있다 (브로드캐스팅 연산). 추가적인 이점으로는 다른 dot call 과 같이 쓴다면 하나의 루프로 융합한다는 것이다 (예: `sin.(cos.(x))`).

또한, 모든 이항 연산자는 점을 찍어사용할 수 있으며, 이는 융합 브로드캐스팅 연산에서 배열(그리고 배열과 스칼라의 조합)에 적용할 수 있다 (예: `z .== sin.(x .* y)`).

참고로, `==` 와 같은 연산자는 전체 배열에 적용되어, 단 하나의 부울 값을 내어놓는다. 원소별 비교를 위해서는 점 `.==`와 같은 점 연산자를 사용하라. (`<`와 같은 비교 연산은, 원소별 연산 `.<`만이 배열에 적용 가능하다.)

또한, `max`를 `a`와 `b`에 원소별로 `broadcast` 하는 `max.(a,b)`와 , `a`의 최대값을 찾는 `maximum(a)`의 차이에 유의하라. `min.(a,b)`와 `minimum(a)` 의 관계도 마찬가지이다.

19.13 브로드캐스팅

행렬의 각 배열을 더하는 것 처럼, 다른 크기의 배열들을 원소별로 이항 연산할 필요가 종종 있다. 이를 비효율적으로 하는 방법은 벡터를 행렬과 같은 크기로 복사하는 것이다:

```
julia> a = rand(2,1); A = rand(2,3);

julia> repeat(a,1,3)+A
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

차원의 크기가 커지면 위 방법은 낭비가 심해지므로, Julia는 broadcast를 제공한다. broadcast는 추가적인 메모리를 사용하지 않으면서, 주어진 한 배열의 차원 중 크기가 1인 차원을 주어진 다른 배열의 해당 차원의 크기와 일치하도록 확장하여 주어진 함수를 원소별로 적용하는 함수이다:

```
julia> broadcast(+, a, A)
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1×2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2×2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

.+ 와 .* 같은 점찍은 연산자는 broadcast 호출과 (아래에 설명할 융합을 제외한다면) 동일하다. 또한 명시적으로 목적지를 지정하는 broadcast!도 있다. (.= 대입을 사용하여 융합하여서도 액세스할 수 있다.) 사실, f.(args...)는 broadcast(f, args...)와 동일하며, 어떤 함수든 점 문법을 통하여 편리하게 브로드캐스팅 할 수 있는 문법을 제공한다. 중첩된 "점 호출" f.(...)은 (.+ 등의 연산자도 포함하여) 하나의 broadcast 호출로 자동으로 융합한다.

추가적으로, broadcast는 배열에 국한되지 않고 (함수 문서 참조) 튜플 또한 지원하며, 배열, 튜플, Ref(Ptr 제외)가 아닌 모든 값은 "스칼라"로 취급한다.

```
julia> convert.(Float32, [1, 2])
2-element Array{Float32,1}:
```

```
 1.0
 2.0

julia> ceil.((UInt8,), [1.2 3.4; 5.6 6.7])
2×2 Array{UInt8,2}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Array{String,1}:
 "1. First"
 "2. Second"
 "3. Third"
```

19.14 구현

Julia에서 기본 배열 타입은 추상 타입인 `AbstractArray{T,N}`이다. `AbstractArray{T,N}`는 차원수 `N`과 원소 타입 `T`로 매개변수화 되어 있다. `AbstractVector`와 `AbstractMatrix`는 일차원과 이차원 배열의 앨리어스(alias)이다. `AbstractArray` 객체에 대한 연산은 기저 스토리지에 독립적인 형태로 고수준의 연산자와 함수를 사용하여 정의된다. 이 연산은 일반적으로 구체적 배열 구현의 폴백(fallback)으로서 정상동작한다.

`AbstractArray` 타입은 배열과 비슷한 모든 것을 포함하며, 이들의 구현은 전통적인 배열과는 차이가 많이 날 수도 있다. 예를 들어, 원소를 저장하지 않고 요청에 따라서 계산할 수도 있다. 다만 모든 구체적인 `AbstractArray{T,N}` 타입은 일반적으로 적어도 (`Int` 튜플을 리턴하는) `size(A)`, `getindex(A,i)`, 그리고 `getindex(A,i1,...,iN)`를 구현해야 한다. 변경 가능한 배열은 `setindex!`도 구현해야 한다. 이러한 연산들은 대략 상수 시간 복잡도, 엄밀히 말해 ⊠(1) 복잡도를 가지도록 구현하는 것이 좋다. 그렇지 않으면 어떤 배열 함수는 생각 이상으로 느릴지도 모른다. 구체적 타입은 `copy`등의 out-of-place 연산에서 유사한 배열을 할당하는데에 쓰일 수 있는 `similar(A,T=eltype(A),dims=size(A))`메소드를 제공해야 한다. `AbstractArray{T,N}`가 내부적으로 어떻게 표현이 되든, `T` 는 정수 인덱싱이 리턴하는 객체(`A` 가 빈 배열이 아닌 경우 `A[1, ..., 1]`)의 타입이며, `N`은 `size`가 리턴하는 튜플의 길이여야 한다. For more details on defining custom `AbstractArray` implementations, see the array interface guide in the interfaces chapter.

`DenseArray` is an abstract subtype of `AbstractArray` intended to include all arrays where elements are stored contiguously in column-major order (see additional notes in Performance Tips). The `Array` type is a specific instance of `DenseArray`; `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Very few operations are implemented specifically for `Array` beyond those that are required for all `AbstractArray`s; much of the array library is implemented in a generic manner that allows all custom arrays to behave similarly.

`SubArray` is a specialization of `AbstractArray` that performs indexing by sharing memory with the original array rather than by copying it. A `SubArray` is created with the `view` function, which is called the same way as `getindex` (with an array and a series of index arguments). The result of `view` looks the same as the result of `getindex`, except

the data is left in place. `view` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly. `@views` 매크로를 표현식이나 코드 블록 앞에 둠으로써, 그 표현식 내의 모든 `array[...]` 슬라이스가 `SubArray` 뷰를 생성하도록 할 수 있다.

`BitArray`s are space-efficient "packed" boolean arrays, which store one bit per boolean value. They can be used similarly to `Array{Bool}` arrays (which store one byte per boolean value), and can be converted to/from the latter via `Array(bitarray)` and `BitArray(array)`, respectively.

A "strided" array is stored in memory with elements laid out in regular offsets such that an instance with a supported `isbits` element type can be passed to external C and Fortran functions that expect this memory layout. Strided arrays must define a `strides(A)` method that returns a tuple of "strides" for each dimension; a provided `stride(A,k)` method accesses the `k`th element within this tuple. Increasing the index of dimension `k` by `1` should increase the index `i` of `getindex(A,i)` by `stride(A,k)`. If a pointer conversion method `Base.unsafe_convert(Ptr{T}, A)` is provided, the memory layout must correspond in the same way to these strides. `DenseArray` is a very specific example of a strided array where the elements are arranged contiguously, thus it provides its subtypes with the approporiate definition of `strides`. More concrete examples can be found within the interface guide for strided arrays. `StridedVector` and `StridedMatrix` are convenient aliases for many of the builtin array types that are considered strided arrays, allowing them to dispatch to select specialized implementations that call highly tuned and optimized BLAS and LAPACK functions using just the pointer and strides.

다음 예시에서는 임시 배열을 만들지 않고 적절한 LAPACK 함수를 차원 크기와 스트라이드를 사용하여 호출하여 큰 배열의 작은 섹션의 QR 분해를 계산한다.

```
julia> a = rand(10, 10)
10×10 Array{Float64,2}:
 0.517515  0.0348206  0.749042   0.0979679  …  0.75984     0.950481   0.579513
 0.901092  0.873479   0.134533   0.0697848     0.0586695   0.193254   0.726898
 0.976808  0.0901881  0.208332   0.920358      0.288535    0.705941   0.337137
 0.657127  0.0317896  0.772837   0.534457      0.0966037   0.700694   0.675999
 0.471777  0.144969   0.0718405  0.0827916     0.527233    0.173132   0.694304
 0.160872  0.455168   0.489254   0.827851   …  0.62226     0.0995456  0.946522
 0.291857  0.769492   0.68043    0.629461      0.727558    0.910796   0.834837
 0.775774  0.700731   0.700177   0.0126213     0.00822304  0.327502   0.955181
 0.9715    0.64354    0.848441   0.241474      0.591611    0.792573   0.194357
 0.646596  0.575456   0.0995212  0.038517      0.709233    0.477657   0.0507231

julia> b = view(a, 2:2:8,2:2:4)
4×2 view(::Array{Float64,2}, 2:2:8, 2:2:4) with eltype Float64:
 0.873479   0.0697848
 0.0317896  0.534457
```

```
 0.455168   0.827851
 0.700731   0.0126213

julia> (q, r) = qr(b);

julia> q
4×4 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.722358    0.227524  -0.247784    -0.604181
 -0.0262896  -0.575919  -0.804227     0.144377
 -0.376419   -0.75072    0.540177    -0.0541979
 -0.579497    0.230151  -0.00552346   0.781782

julia> r
2×2 Array{Float64,2}:
 -1.20921  -0.383393
  0.0      -0.910506
```

Chapter 20

Missing Values

Julia provides support for representing missing values in the statistical sense, that is for situations where no value is available for a variable in an observation, but a valid value theoretically exists. Missing values are represented via the `missing` object, which is the singleton instance of the type `Missing`. **`missing`** is equivalent to `NULL` in SQL and `NA` in R, and behaves like them in most situations.

20.1 Propagation of Missing Values

The behavior of **`missing`** values follows one basic rule: **`missing`** values propagate automatically when passed to standard operators and functions, in particular mathematical functions. Uncertainty about the value of one of the operands induces uncertainty about the result. In practice, this means an operation involving a **`missing`** value generally returns **`missing`**

```
julia> missing + 1
missing

julia> "a" * missing
missing

julia> abs(missing)
missing
```

As **`missing`** is a normal Julia object, this propagation rule only works for functions which have opted in to implement this behavior. This can be achieved either via a specific method defined for arguments of type **`Missing`**, or simply by accepting arguments of this type, and passing them to functions which propagate them (like standard operators). Packages should consider whether it makes sense to propagate missing values when defining new functions, and

define methods appropriately if that is the case. Passing a **missing** value to a function for which no method accepting arguments of type **Missing** is defined throws a MethodError, just like for any other type.

20.2 Equality and Comparison Operators

Standard equality and comparison operators follow the propagation rule presented above: if any of the operands is **missing**, the result is **missing**. Here are a few examples

```
julia> missing == 1
missing

julia> missing == missing
missing

julia> missing < 1
missing

julia> 2 >= missing
missing
```

In particular, note that **missing == missing** returns **missing**, so **==** cannot be used to test whether a value is missing. To test whether x is **missing**, use ismissing(x).

Special comparison operators isequal and **===** are exceptions to the propagation rule: they always return a **Bool** value, even in the presence of **missing** values, considering **missing** as equal to **missing** and as different from any other value. They can therefore be used to test whether a value is **missing**

```
julia> missing === 1
false

julia> isequal(missing, 1)
false

julia> missing === missing
true

julia> isequal(missing, missing)
true
```

The isless operator is another exception: missing is considered as greater than any other value. This operator is used by sort, which therefore places missing values after all other values.

```
julia> isless(1, missing)
true

julia> isless(missing, Inf)
false

julia> isless(missing, missing)
false
```

20.3 Logical operators

Logical (or boolean) operators |, & and xor are another special case, as they only propagate missing values when it is logically required. For these operators, whether or not the result is uncertain depends on the particular operation, following the well-established rules of three-valued logic which are also implemented by NULL in SQL and NA in R. This abstract definition actually corresponds to a relatively natural behavior which is best explained via concrete examples.

Let us illustrate this principle with the logical "or" operator |. Following the rules of boolean logic, if one of the operands is true, the value of the other operand does not have an influence on the result, which will always be true

```
julia> true | true
true

julia> true | false
true

julia> false | true
true
```

Based on this observation, we can conclude that if one of the operands is true and the other missing, we know that the result is true in spite of the uncertainty about the actual value of one of the operands. If we had been able to observe the actual value of the second operand, it could only be true or false, and in both cases the result would be true. Therefore, in this particular case, missingness does not propagate

```
julia> true | missing
true
```

```
julia> missing ¦ true
true
```

On the contrary, if one of the operands is `false`, the result could be either `true` or `false` depending on the value of the other operand. Therefore, if that operand is `missing`, the result has to be `missing` too

```
julia> false ¦ true
true

julia> true ¦ false
true

julia> false ¦ false
false

julia> false ¦ missing
missing

julia> missing ¦ false
missing
```

The behavior of the logical "and" operator `&` is similar to that of the `¦` operator, with the difference that missingness does not propagate when one of the operands is `false`. For example, when that is the case of the first operand

```
julia> false & false
false

julia> false & true
false

julia> false & missing
false
```

On the other hand, missingness propagates when one of the operands is `true`, for example the first one

```
julia> true & true
true
```

```
julia> true & false
false

julia> true & missing
missing
```

Finally, the "exclusive or" logical operator `xor` always propagates **`missing`** values, since both operands always have an effect on the result. Also note that the negation operator `!` returns **`missing`** when the operand is **`missing`** just like other unary operators.

20.4 Control Flow and Short-Circuiting Operators

Control flow operators including `if`, `while` and the ternary operator `x ? y : z` do not allow for missing values. This is because of the uncertainty about whether the actual value would be **`true`** or **`false`** if we could observe it, which implies that we do not know how the program should behave. A `TypeError` is thrown as soon as a **`missing`** value is encountered in this context

```
julia> if missing
           println("here")
       end
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

For the same reason, contrary to logical operators presented above, the short-circuiting boolean operators `&&` and `||` do not allow for **`missing`** values in situations where the value of the operand determines whether the next operand is evaluated or not. For example

```
julia> missing || false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> true && missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

On the other hand, no error is thrown when the result can be determined without the **`missing`** values. This is the case when the code short-circuits before evaluating the **`missing`** operand, and when the **`missing`** operand is the last one

```
julia> true && missing
missing

julia> false && missing
false
```

20.5 Arrays With Missing Values

Arrays containing missing values can be created like other arrays

```
julia> [1, missing]
2-element Array{Union{Missing, Int64},1}:
 1
  missing
```

As this example shows, the element type of such arrays is `Union{Missing, T}`, with `T` the type of the non-missing values. This simply reflects the fact that array entries can be either of type `T` (here, `Int64`) or of type `Missing`. This kind of array uses an efficient memory storage equivalent to an `Array{T}` holding the actual values combined with an `Array{UInt8}` indicating the type of the entry (i.e. whether it is `Missing` or `T`).

Arrays allowing for missing values can be constructed with the standard syntax. Use `Array{Union{Missing, T}}(missing, dims)` to create arrays filled with missing values:

```
julia> Array{Union{Missing, String}}(missing, 2, 3)
2×3 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

An array allowing for `missing` values but which does not contain any such value can be converted back to an array which does not allow for missing values using `convert`. If the array contains `missing` values, a `MethodError` is thrown during conversion

```
julia> x = Union{Missing, String}["a", "b"]
2-element Array{Union{Missing, String},1}:
 "a"
 "b"

julia> convert(Array{String}, x)
2-element Array{String,1}:
```

```
 "a"
 "b"

julia> y = Union{Missing, String}[missing, "b"]
2-element Array{Union{Missing, String},1}:
 missing
 "b"

julia> convert(Array{String}, y)
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type String
```

20.6 Skipping Missing Values

Since **missing** values propagate with standard mathematical operators, reduction functions return **missing** when called on arrays which contain missing values

```
julia> sum([1, missing])
missing
```

In this situation, use the skipmissing function to skip missing values

```
julia> sum(skipmissing([1, missing]))
1
```

This convenience function returns an iterator which filters out **missing** values efficiently. It can therefore be used with any function which supports iterators

```
julia> x = skipmissing([3, missing, 2, 1])
Base.SkipMissing{Array{Union{Missing, Int64},1}}(Union{Missing, Int64}[3, missing, 2, 1])

julia> maximum(x)
3

julia> mean(x)
2.0

julia> mapreduce(sqrt, +, x)
4.146264369941973
```

Objects created by calling **skipmissing** on an array can be indexed using indices from the parent array. Indices corresponding to missing values are not valid for these objects and an error is thrown when trying to use them (they are also skipped by **keys** and **eachindex**)

```
julia> x[1]
3

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]
```

This allows functions which operate on indices to work in combination with **skipmissing**. This is notably the case for search and find functions, which return indices valid for the object returned by **skipmissing** which are also the indices of the matching entries in the parent array

```
julia> findall(==(1), x)
1-element Array{Int64,1}:
 4

julia> findfirst(!iszero, x)
1

julia> argmax(x)
1
```

Use collect to extract non-**missing** values and store them in an array

```
julia> collect(x)
3-element Array{Int64,1}:
 3
 2
 1
```

20.7 Logical Operations on Arrays

The three-valued logic described above for logical operators is also used by logical functions applied to arrays. Thus, array equality tests using the == operator return **missing** whenever the result cannot be determined without knowing the actual value of the **missing** entry. In practice, this means that **missing** is returned if all non-missing values of the compared arrays are equal, but one or both arrays contain missing values (possibly at different positions)

```
julia> [1, missing] == [2, missing]
false

julia> [1, missing] == [1, missing]
missing

julia> [1, 2, missing] == [1, missing, 2]
missing
```

As for single values, use `isequal` to treat `missing` values as equal to other `missing` values but different from non-missing values

```
julia> isequal([1, missing], [1, missing])
true

julia> isequal([1, 2, missing], [1, missing, 2])
false
```

Functions `any` and `all` also follow the rules of three-valued logic, returning `missing` when the result cannot be determined

```
julia> all([true, missing])
missing

julia> all([false, missing])
false

julia> any([true, missing])
true

julia> any([false, missing])
missing
```

Chapter 21

Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets. This interface, though asynchronous at the system level, is presented in a synchronous manner to the programmer and it is usually unnecessary to think about the underlying asynchronous operation. This is achieved by making heavy use of Julia cooperative threading (coroutine) functionality.

21.1 Basic Stream I/O

All Julia streams expose at least a `read` and a `write` method, taking the stream as their first argument, e.g.:

```
julia> write(stdout, "Hello World");  # suppress return value 11 with ;
Hello World
julia> read(stdin, Char)


'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Note that `write` returns 11, the number of bytes (in `"Hello World"`) written to `stdout`, but this return value is suppressed with the `;`.

Here Enter was pressed again so that Julia would read the newline. Now, as you can see from this example, `write` takes the data to write as its second argument, while `read` takes the type of the data to be read as the second argument.

For example, to read a simple byte array, we could do:

```
julia> x = zeros(UInt8, 4)
4-element Array{UInt8,1}:
 0x00
 0x00
```

```
 0x00
 0x00

julia> read!(stdin, x)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```
julia> read(stdin, 4)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

or if we had wanted to read the entire line instead:

```
julia> readline(stdin)
abcd
"abcd"
```

Note that depending on your terminal settings, your TTY may be line buffered and might thus require an additional enter before the data is sent to Julia.

To read every line from `stdin` you can use `eachline`:

```
for line in eachline(stdin)
    print("Found $line")
end
```

or `read` if you wanted to read by character instead:

```
while !eof(stdin)
    x = read(stdin, Char)
    println("Found: $x")
end
```

21.2 Text I/O

Note that the `write` method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

```
julia> write(stdout, 0x61);  # suppress return value 1 with ;
a
```

Note that `a` is written to `stdout` by the `write` function and that the returned value is `1` (since `0x61` is one byte).

For text I/O, use the `print` or `show` methods, depending on your needs (see the documentation for these two methods for a detailed discussion of the difference between them):

```
julia> print(stdout, 0x61)
97
```

See Custom pretty-printing for more information on how to implement display methods for custom types.

21.3 IO Output Contextual Properties

Sometimes IO output can benefit from the ability to pass contextual information into show methods. The `IOContext` object provides this framework for associating arbitrary metadata with an IO object. For example, `:compact => true` adds a hinting parameter to the IO object that the invoked show method should print a shorter output (if applicable). See the `IOContext` documentation for a list of common properties.

21.4 Working with Files

Like many other environments, Julia has an `open` function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example, if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)
```

```
julia> readlines(f)
1-element Array{String,1}:
 "Hello, World!"
```

If you want to write to a file, you can open it with the write (`"w"`) flag:

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

If you examine the contents of `hello.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IOStream` must be closed before the write is actually flushed to disk:

```
julia> close(f)
```

Examining `hello.txt` again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of `open` which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again. For example, given a function:

```
function read_and_capitalize(f::IOStream)
    return uppercase(read(f, String))
end
```

You can call:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

to open `hello.txt`, call `read_and_capitalize` on it, close `hello.txt` and return the capitalized contents.

To avoid even having to define a named function, you can use the `do` syntax, which creates an anonymous function on the fly:

```
julia> open("hello.txt") do f
           uppercase(read(f, String))
       end
"HELLO AGAIN."
```

21.5 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. This functionality is in a standard library package called `Sockets`. Let's first create a simple server:

```
julia> using Sockets

julia> @async begin
           server = listen(2000)
           while true
               sock = accept(server)
               println("Hello World\n")
           end
       end
Task (runnable) @0x00007fd31dc11ae0
```

To those familiar with the Unix socket API, the method names will feel familiar, though their usage is somewhat simpler than the raw Unix socket API. The first call to `listen` will create a server waiting for incoming connections on the specified port (2000) in this case. The same function may also be used to create various other kinds of servers:

```
julia> listen(2000) # Listens on localhost:2000 (IPv4)
Sockets.TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
Sockets.TCPServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
Sockets.TCPServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
Sockets.TCPServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
Sockets.TCPServer(active)
```

```
julia> listen("testsocket") # Listens on a UNIX domain socket
Sockets.PipeServer(active)

julia> listen("\\\\.\\pipe\\testsocket") # Listens on a Windows named pipe
Sockets.PipeServer(active)
```

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix (`\\.\pipe\`) uniquely identifies the file type. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the `accept` and `connect` methods. The `accept` method retrieves a connection to the client that is connecting on the server we just created, while the `connect` function connects to a server using the specified method. The `connect` function takes the same arguments as `listen`, so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to `connect` as you did to listen to establish the connection. So let's try that out (after having created the server above):

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called `connect`, we connect to the server we had just created. Meanwhile, the accept function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry about callbacks or even making sure that the server gets to run. When we called `connect` the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

```
julia> @async begin
           server = listen(2001)
           while true
               sock = accept(server)
               @async while isopen(sock)
                   write(sock, readline(sock, keep=true))
```

```
            end
        end
    end
Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> @async while isopen(clientside)
        write(stdout, readline(clientside, keep=true))
    end
Task (runnable) @0x00007fd31dc11870

julia> println(clientside,"Hello World from the Echo Server")
Hello World from the Echo Server
```

As with other streams, use `close` to disconnect the socket:

```
julia> close(clientside)
```

21.6 Resolving IP Addresses

One of the `connect` methods that does not follow the `listen` methods is `connect(host::String,port)`, which will attempt to connect to the host given by the `host` parameter on the port given by the `port` parameter. It allows you to do things like:

```
julia> connect("google.com", 80)
TCPSocket(RawFD(30) open, 0 bytes waiting)
```

At the base of this functionality is `getaddrinfo`, which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

Chapter 22

Parallel Computing

For newcomers to multi-threading and parallel computing it can be useful to first appreciate the different levels of parallelism offered by Julia. We can divide them in three main categories :

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

We will first consider Julia Tasks (aka Coroutines) and other modules that rely on the Julia runtime library, that allow us to suspend and resume computations with full control of inter-`Tasks` communication without having to manually interface with the operating system's scheduler. Julia also supports communication between `Tasks` through operations like `wait` and `fetch`. Communication and data synchronization is managed through `Channel`s, which are the conduits that provide inter-`Tasks` communication.

Julia also supports experimental multi-threading, where execution is forked and an anonymous function is run across all threads. Known as the fork-join approach, parallel threads execute independently, and must ultimately be joined in Julia's main thread to allow serial execution to continue. Multi-threading is supported using the `Base.Threads` module that is still considered experimental, as Julia is not yet fully thread-safe. In particular segfaults seem to occur during I/O operations and task switching. As an up-to-date reference, keep an eye on the issue tracker. Multi-Threading should only be used if you take into consideration global variables, locks and atomics, all of which are explained later.

In the end we will present Julia's approach to distributed and parallel computing. With scientific computing in mind, Julia natively implements interfaces to distribute a process across multiple cores or machines. Also we will mention useful external packages for distributed programming like `MPI.jl` and `DistributedArrays.jl`.

Chapter 23

Coroutines

Julia's parallel programming platform uses Tasks (aka Coroutines) to switch among multiple computations. To express an order of execution between lightweight threads communication primitives are necessary. Julia offers `Channel(func::Function, ctype=Any, csize=0, taskref=nothing)` that creates a new task from `func`, binds it to a new channel of type `ctype` and size `csize` and schedule the task. `Channels` can serve as a way to communicate between tasks, as `Channel{T}(sz::Int)` creates a buffered channel of type `T` and size `sz`. Whenever code performs a communication operation like `fetch` or `wait`, the current task is suspended and a scheduler picks another task to run. A task is restarted when the event it is waiting for completes.

For many problems, it is not necessary to think about tasks directly. However, they can be used to wait for multiple events at the same time, which provides for dynamic scheduling. In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish. This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processes only when they finish their current tasks.

23.1 Channels

The section on `Task`s in 제어 흐름 discussed the execution of multiple functions in a co-operative manner. `Channel`s can be quite useful to pass data between running tasks, particularly those involving I/O operations.

Examples of operations involving I/O include reading/writing to files, accessing web services, executing external programs, etc. In all these cases, overall execution time can be improved if other tasks can be run while a file is being read, or while waiting for an external service/program to complete.

A channel can be visualized as a pipe, i.e., it has a write end and a read end :

- Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.
- Multiple readers in different tasks can read data concurrently via `take!` calls.

- As an example:

```
# Given Channels c1 and c2,
c1 = Channel(32)
c2 = Channel(32)

# and a function `foo` which reads items from c1, processes the item read
# and writes a result to c2,
function foo()
    while true
        data = take!(c1)
        [...]               # process data
        put!(c2, result)    # write out result
    end
end

# we can schedule `n` instances of `foo` to be active concurrently.
for _ in 1:n
    @async foo()
end
```

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of type `T`. If the type is not specified, the channel can hold objects of any type. `sz` refers to the maximum number of elements that can be held in the channel at any time. For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type. A `Channel{MyType}(64)` can hold up to 64 objects of `MyType` at any time.

- If a `Channel` is empty, readers (on a `take!` call) will block until data is available.

- If a `Channel` is full, writers (on a `put!` call) will block until space becomes available.

- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.

- A `Channel` is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls. `close` closes a `Channel`. On a closed `Channel`, `put!` will fail. For example:

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);
```

```
julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException("Channel is closed.",:closed)
Stacktrace:
[...]
```

- `take!` and `fetch` (which retrieves but does not remove the value) on a closed channel successfully return any existing values until it is emptied. Continuing the above example:

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1

julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
ERROR: InvalidStateException("Channel is closed.",:closed)
Stacktrace:
[...]
```

A `Channel` can be used as an iterable object in a `for` loop, in which case the loop runs as long as the `Channel` has data or is open. The loop variable takes on all values added to the `Channel`. The `for` loop is terminated once the `Channel` is closed and emptied.

For example, the following would cause the `for` loop to wait for more data:

```
julia> c = Channel{Int}(10);

julia> foreach(i->put!(c, i), 1:3) # add a few entries

julia> data = [i for i in c]
```

while this will return after reading all data:

```
julia> c = Channel{Int}(10);

julia> foreach(i->put!(c, i), 1:3); # add a few entries
```

```
julia> close(c);                    # `for` loops can exit

julia> data = [i for i in c]
3-element Array{Int64,1}:
 1
 2
 3
```

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single `jobs` channel. Jobs, identified by an id (`job_id`), are written to the channel. Each task in this simulation reads a `job_id`, waits for a random amount of time and writes back a tuple of `job_id` and the simulated time to the results channel. Finally all the `results` are printed out.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()
           for job_id in jobs
               exec_time = rand()
               sleep(exec_time)                # simulates elapsed time doing actual work
                                               # typically performed externally.
               put!(results, (job_id, exec_time))
           end
       end;

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> @async make_jobs(n); # feed the jobs channel with "n" jobs

julia> for i in 1:4 # start 4 tasks to process requests in parallel
           @async do_work()
       end
```

```
julia> @elapsed while n > 0 # print out results
           job_id, exec_time = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds")
           global n = n - 1
       end
4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
9 finished in 0.36 seconds
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

The current version of Julia multiplexes all tasks onto a single OS thread. Thus, while tasks involving I/O operations benefit from parallel execution, compute bound tasks are effectively executed sequentially on a single OS thread. Future versions of Julia may support scheduling of tasks on multiple threads, in which case compute bound tasks will see benefits of parallel execution too.

Chapter 24

Multi-Threading (Experimental)

In addition to tasks Julia forwards natively supports multi-threading. Note that this section is experimental and the interfaces may change in the future.

24.1 Setup

By default, Julia starts up with a single thread of execution. This can be verified by using the command `Threads.nthreads()`:

```
julia> Threads.nthreads()
1
```

The number of threads Julia starts up with is controlled by an environment variable called `JULIA_NUM_THREADS`. Now, let's start up Julia with 4 threads:

```
export JULIA_NUM_THREADS=4
```

(The above command works on bourne shells on Linux and OSX. Note that if you're using a C shell on these platforms, you should use the keyword `set` instead of `export`. If you're on Windows, start up the command line in the location of `julia.exe` and use `set` instead of `export`.)

Let's verify there are 4 threads at our disposal.

```
julia> Threads.nthreads()
4
```

But we are currently on the master thread. To check, we use the function `Threads.threadid`

```
julia> Threads.threadid()
1
```

24.2 The `@threads` Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
julia> a = zeros(10)
10-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the `Threads.@threads` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
julia> Threads.@threads for i = 1:10
           a[i] = Threads.threadid()
       end
```

The iteration space is split amongst the threads, after which each thread writes its thread ID to its assigned locations:

```
julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
```

```
 4.0
 4.0
```

Note that `Threads.@threads` does not have an optional reduction parameter like `@distributed`.

24.3 Atomic Operations

Julia supports accessing and modifying values atomically, that is, in a thread-safe way to avoid race conditions. A value (which must be of a primitive type) can be wrapped as `Threads.Atomic` to indicate it must be accessed in this way. Here we can see an example:

```
julia> i = Threads.Atomic{Int}(0);

julia> ids = zeros(4);

julia> old_is = zeros(4);

julia> Threads.@threads for id in 1:4
           old_is[id] = Threads.atomic_add!(i, id)
           ids[id] = id
       end

julia> old_is
4-element Array{Float64,1}:
 0.0
 1.0
 7.0
 3.0

julia> ids
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
```

Had we tried to do the addition without the atomic tag, we might have gotten the wrong answer due to a race condition. An example of what would happen if we didn't avoid the race:

```
julia> using Base.Threads

julia> nthreads()
4

julia> acc = Ref(0)
Base.RefValue{Int64}(0)

julia> @threads for i in 1:1000
          acc[] += 1
       end

julia> acc[]
926

julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)

julia> @threads for i in 1:1000
          atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

> Note
>
> Not all primitive types can be wrapped in an `Atomic` tag. Supported types are `Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `Float16`, `Float32`, and `Float64`. Additionally, `Int128` and `UInt128` are not supported on AAarch32 and ppc64le.

24.4 Side effects and mutable function arguments

When using multi-threading we have to be careful when using functions that are not pure as we might get a wrong answer. For instance functions that have their name ending with `!` by convention modify their arguments and thus are not pure. However, there are functions that have side effects and their name does not end with `!`. For instance `findfirst(regex, str)` mutates its `regex` argument or `rand()` changes `Base.GLOBAL_RNG` :

```
julia> using Base.Threads
```

```
julia> nthreads()
4

julia> function f()
           s = repeat(["123", "213", "231"], outer=1000)
           x = similar(s, Int)
           rx = r"1"
           @threads for i in 1:3000
               x[i] = findfirst(rx, s[i]).start
           end
           count(v -> v == 1, x)
       end
f (generic function with 1 method)

julia> f() # the correct result is 1000
1017

julia> function g()
           a = zeros(1000)
           @threads for i in 1:1000
               a[i] = rand()
           end
           length(unique(a))
       end
g (generic function with 1 method)

julia> Random.seed!(1); g() # the result for a single thread is 1000
781
```

In such cases one should redesign the code to avoid the possibility of a race condition or use synchronization primitives.

For example in order to fix `findfirst` example above one needs to have a separate copy of `rx` variable for each thread:

```
julia> function f_fix()
            s = repeat(["123", "213", "231"], outer=1000)
            x = similar(s, Int)
            rx = [Regex("1") for i in 1:nthreads()]
            @threads for i in 1:3000
                x[i] = findfirst(rx[threadid()], s[i]).start
```

```
            end
            count(v -> v == 1, x)
        end
f_fix (generic function with 1 method)

julia> f_fix()
1000
```

We now use `Regex("1")` instead of `r"1"` to make sure that Julia creates separate instances of `Regex` object for each entry of `rx` vector.

The case of `rand` is a bit more complex as we have to ensure that each thread uses non-overlapping pseudorandom number sequences. This can be simply ensured by using `Future.randjump` function:

```
julia> using Random; import Future

julia> function g_fix(r)
           a = zeros(1000)
           @threads for i in 1:1000
               a[i] = rand(r[threadid()])
           end
           length(unique(a))
       end
g_fix (generic function with 1 method)

julia>  r = let m = MersenneTwister(1)
               [m; accumulate(Future.randjump, fill(big(10)^20, nthreads()-1), init=m)]
           end;

julia> g_fix(r)
1000
```

We pass the `r` vector to `g_fix` as generating several RGNs is an expensive operation so we do not want to repeat it every time we run the function.

24.5 @threadcall (Experimental)

All I/O tasks, timers, REPL commands, etc are multiplexed onto a single OS thread via an event loop. A patched version of libuv (http://docs.libuv.org/en/v1.x/) provides this functionality. Yield points provide for co-operatively

scheduling multiple tasks onto the same OS thread. I/O tasks and timers yield implicitly while waiting for the event to occur. Calling `yield` explicitly allows for other tasks to be scheduled.

Thus, a task executing a `ccall` effectively prevents the Julia scheduler from executing any other tasks till the call returns. This is true for all calls into external libraries. Exceptions are calls into custom C code that call back into Julia (which may then yield) or C code that calls `jl_yield()` (C equivalent of `yield`).

Note that while Julia code runs on a single thread (by default), libraries used by Julia may launch their own internal threads. For example, the BLAS library may start as many threads as there are cores on a machine.

The `@threadcall` macro addresses scenarios where we do not want a `ccall` to block the main Julia event loop. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable `UV_THREADPOOL_SIZE`. While waiting for a free thread, and during function execution once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that `@threadcall` does not return till the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia, as it will segfault.

`@threadcall` may be removed/changed in future versions of Julia.

Chapter 25

Multi-Core or Distributed Processing

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI [1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Distributed programming in Julia is built on two primitives: remote references and remote calls. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: `Future` and `RemoteChannel`.

A remote call returns a `Future` to its result. Remote calls return immediately; the process that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling `wait` on the returned `Future`, and you can obtain the full value of the result using `fetch`.

On the other hand, `RemoteChannel` s are rewritable. For example, multiple processes can co-ordinate their processing by referencing the same remote `Channel`.

Each process has an associated identifier. The process providing the interactive Julia prompt always has an `id` equal to 1. The processes used by default for parallel operations are referred to as "workers". When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

Let's try this out. Starting with `julia -p n` provides `n` worker processes on the local machine. Generally it makes sense for `n` to equal the number of CPU threads (logical cores) on the machine. Note that the `-p` argument implicitly loads module `Distributed`.

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, nothing)

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

The first argument to `remotecall` is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but `remotecall` is considered a low-level interface providing finer control. The second argument to `remotecall` is the `id` of the process that will do the work, and the remaining arguments will be passed to the function being called.

As you can see, in the first line we asked process 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two futures, `r` and `s`. The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function `remotecall_fetch` exists for this purpose. It is equivalent to `fetch(remotecall(...))` but is more efficient.

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.18526337335308085
```

Remember that `getindex(r,1,1)` is equivalent to `r[1,1]`, so this call fetches the first element of the future `r`.

To make things easier, the symbol `:any` can be passed to `[@spawnat]`, which picks where to do the operation for you:

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
Future(3, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

Note that we used `1 .+ fetch(r)` instead of `1 .+ r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the process doing the addition. In this case, `@spawnat` is smart enough to perform the computation on the process that owns `r`, so the `fetch` will be a no-op (no work is done).

(It is worth noting that `@spawnat` is not built-in but defined in Julia as a macro. It is possible to define your own such constructs.)

An important thing to remember is that, once fetched, a `Future` will cache its value locally. Further `fetch` calls do not entail a network hop. Once all referencing `Future`s have fetched, the remote stored value is deleted.

`@async` is similar to `@spawnat`, but only runs tasks on the local process. We use it to create a "feeder" task for each process. Each task picks the next index that needs to be computed, then waits for its process to finish, then repeats until we run out of indices. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `@sync` block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. As for v0.7 and beyond, the feeder tasks are able to share state via `nextidx` because they all run on the same process. Even if `Tasks` are scheduled cooperatively, locking may still be required in some contexts, as in asynchronous I/O. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch` is called. This is the current state of implementation and it may change for future Julia versions, as it is intended to make it possible to run up to N `Tasks` on M `Process`, aka M:N Threading. Then a lock acquiring₩releasing model for `nextidx` will be needed, as it is not safe to let multiple processes read-write a resource at the same time.

25.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

```
julia> function rand2(dims...)
           return 2*rand(dims...)
       end

julia> rand2(2,2)
2×2 Array{Float64,2}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawnat :any rand2(2,2))
ERROR: RemoteException(2, CapturedException(UndefVarError(Symbol("#rand2"))
Stacktrace:
[...]
```

Process 1 knew about the function `rand2`, but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, `DummyModule.jl`, containing the following code:

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

In order to refer to `MyType` across all processes, `DummyModule.jl` needs to be loaded on every process. Calling `include("DummyModule.jl")` loads it only on a single process. To load it on every process, use the `@everywhere` macro (starting Julia with `julia -p 2`):

```
julia> @everywhere include("DummyModule.jl")
loaded
```

```
      From worker 3:    loaded
      From worker 2:    loaded
```

As usual, this does not bring `DummyModule` into scope on any of the process, which requires `using` or `import`. Moreover, when `DummyModule` is brought into scope on one process, it is not on any other:

```
julia> using .DummyModule

julia> MyType(7)
MyType(7)

julia> fetch(@spawnat 2 MyType(7))
ERROR: On worker 2:
UndefVarError: MyType not defined
⋮

julia> fetch(@spawnat 2 DummyModule.MyType(7))
MyType(7)
```

However, it's still possible, for instance, to send a `MyType` to a process which has loaded `DummyModule` even if it's not in scope:

```
julia> put!(RemoteChannel(2), MyType(7))
RemoteChannel{Channel{Any}}(2, 1, 13)
```

A file can also be preloaded on multiple processes at startup with the `-L` flag, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

The Julia process running the driver script in the example above has an `id` equal to 1, just like a process providing an interactive prompt.

Finally, if `DummyModule.jl` is not a standalone file but a package, then `using DummyModule` will load `DummyModule.jl` on all processes, but only bring it into scope on the process where `using` was called.

25.2 Starting and managing worker processes

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the `-p` option as shown above.
- A cluster spanning machines using the `--machine-file` option. This uses a passwordless `ssh` login to start Julia worker processes (from the same path as the current host) on the specified machines.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

```
julia> using Distributed

julia> addprocs(2)
2-element Array{Int64,1}:
 2
 3
```

Module `Distributed` must be explicitly loaded on the master process before invoking `addprocs`. It is automatically made available on the worker processes.

Note that workers do not run a `~/.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

Other types of clusters can be supported by writing your own custom `ClusterManager`, as described below in the ClusterManagers section.

25.3 Data Movement

Sending messages and moving data constitute most of the overhead in a distributed program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various distributed programming constructs.

`fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `@spawnat` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

Method 1:

```
julia> A = rand(1000,1000);

julia> Bref = @spawnat :any A^2;
```

```
[...]

julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawnat :any rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawnat`. In the first method, a random matrix is constructed locally, then sent to another process where it is squared. In the second method, a random matrix is both constructed and squared on another process. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and likely some measurement. For example, if the first process needs matrix `A` then the first method might be better. Or, if computing `A` is expensive and only the current process has it, then moving it to another process might be unavoidable. Or, if the current process has very little to do between the `@spawnat` and `fetch(Bref)`, it might be better to eliminate the parallelism altogether. Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawnat` statement just for this step.

25.4 Global variables

Expressions executed remotely via `@spawnat`, or closures specified for remote execution using `remotecall` may refer to global variables. Global bindings under module `Main` are treated a little differently compared to global bindings in other modules. Consider the following code snippet:

```
A = rand(10,10)
remotecall_fetch(()->sum(A), 2)
```

In this case `sum` MUST be defined in the remote process. Note that `A` is a global variable defined in the local workspace. Worker 2 does not have a variable called `A` under `Main`. The act of shipping the closure `()->sum(A)` to worker 2 results in `Main.A` being defined on 2. `Main.A` continues to exist on worker 2 even after the call `remotecall_fetch` returns. Remote calls with embedded global references (under `Main` module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.
- Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes. For example:

```
A = rand(10,10)
remotecall_fetch(()->sum(A), 2) # worker 2
A = rand(10,10)
remotecall_fetch(()->sum(A), 3) # worker 3
A = nothing
```

  Executing the above snippet results in `Main.A` on worker 2 having a different value from `Main.A` on worker 3, while the value of `Main.A` on node 1 is set to `nothing`.

As you may have realized, while memory associated with globals may be collected when they are reassigned on the master, no such action is taken on the workers as the bindings continue to be valid. `clear!` can be used to manually reassign specific globals on remote nodes to `nothing` once they are no longer required. This will release any memory associated with them as part of a regular garbage collection cycle.

Thus programs should be careful referencing globals in remote calls. In fact, it is preferable to avoid them altogether if possible. If you must reference globals, consider using `let` blocks to localize global variables.

For example:

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);

julia> B = rand(10,10);

julia> let B = B
           remotecall_fetch(()->B, 2)
       end;

julia> @fetchfrom 2 InteractiveUtils.varinfo()
name           size summary
———————— ———————— ———————————————————
A         800 bytes 10×10 Array{Float64,2}
Base                Module
```

```
Core                Module
Main                Module
```

As can be seen, global variable `A` is defined on worker 2, but `B` is captured as a local variable and hence a binding for `B` does not exist on worker 2.

25.5 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a Monte Carlo simulation, where multiple processes can handle independent simulation trials simultaneously. We can use `@spawnat` to flip coins on two processes. First, write the following function in `count_heads.jl`:

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

The function `count_heads` simply adds together `n` random bits. Here is how we can perform some trials on two machines, and add together the results:

```
julia> @everywhere include_string(Main, $(read("count_heads.jl", String)), "count_heads.jl")

julia> a = @spawnat :any count_heads(100000000)
Future(2, 1, 6, nothing)

julia> b = @spawnat :any count_heads(100000000)
Future(3, 1, 7, nothing)

julia> fetch(a)+fetch(b)
100001564
```

This example demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processes, and then their results are combined using some function. The combination process is called a reduction, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern `x = f(x,v[i])`, where

`x` is the accumulator, `f` is the reduction function, and the `v[i]` are the elements being reduced. It is desirable for `f` to be associative, so that it does not matter what order the operations are performed in.

Notice that our use of this pattern with `count_heads` can be generalized. We used two explicit `@spawnat` statements, which limits the parallelism to two processes. To run on any number of processes, we can use a parallel for loop, running in distributed memory, which can be written in Julia using `@distributed` like this:

```
nheads = @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end
```

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case `(+)`). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```
a = zeros(100000)
@distributed for i = 1:100000
    a[i] = i
end
```

This code will not initialize all of `a`, since each process will have a separate copy of it. Parallel for loops like these must be avoided. Fortunately, Shared Arrays can be used to get around this limitation:

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)
@distributed (+) for i = 1:100000
    f(a[rand(1:end)])
end
```

Here each iteration applies `f` to a randomly-chosen sample from a vector `a` shared by all processes.

As you could see, the reduction operator can be omitted if it is not needed. In that case, the loop executes asynchronously, i.e. it spawns independent tasks on all available workers and returns an array of `Future` immediately without waiting for completion. The caller can wait for the `Future` completions at a later point by calling `fetch` on them, or wait for completion at the end of the loop by prefixing it with `@sync`, like `@sync @distributed for`.

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called parallel map, implemented in Julia as the `pmap` function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];

julia> pmap(svdvals, M);
```

Julia's `pmap` is designed for the case where each function call does a large amount of work. In contrast, `@distributed for` can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both `pmap` and `@distributed for` for the parallel computation. In case of `@distributed for`, the final reduction is done on the calling process.

25.6 Remote References and AbstractChannels

Remote references always refer to an implementation of an `AbstractChannel`.

A concrete implementation of an `AbstractChannel` (like `Channel`), is required to implement `put!`, `take!`, `fetch`, `isready` and `wait`. The remote object referred to by a `Future` is stored in a `Channel{Any}(1)`, i.e., a `Channel` of size 1 capable of holding objects of `Any` type.

`RemoteChannel`, which is rewritable, can point to any type and size of channels, or any other implementation of an `AbstractChannel`.

The constructor `RemoteChannel(f::Function, pid)()` allows us to construct references to channels holding more than one value of a specific type. `f` is a function executed on `pid` and it must return an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10. The channel exists on worker `pid`.

Methods `put!`, `take!`, `fetch`, `isready` and `wait` on a `RemoteChannel` are proxied onto the backing store on the remote process.

`RemoteChannel` can thus be used to refer to user implemented `AbstractChannel` objects. A simple example of this is provided in `dictchannel.jl` in the Examples repository, which uses a dictionary as its remote store.

25.7 Channels and RemoteChannels

- A `Channel` is local to a process. Worker 2 cannot directly refer to a `Channel` on worker 3 and vice-versa. A `RemoteChannel`, however, can put and take values across workers.
- A `RemoteChannel` can be thought of as a handle to a `Channel`.
- The process id, `pid`, associated with a `RemoteChannel` identifies the process where the backing store, i.e., the backing `Channel` exists.
- Any process with a reference to a `RemoteChannel` can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a `RemoteChannel` is associated with.
- Serializing a `Channel` also serializes any data present in the channel. Deserializing it therefore effectively makes a copy of the original object.
- On the other hand, serializing a `RemoteChannel` only involves the serialization of an identifier that identifies the location and instance of `Channel` referred to by the handle. A deserialized `RemoteChannel` object (on any worker), therefore also points to the same backing store as the original.

The channels example from above can be modified for interprocess communication, as shown below.

We start 4 workers to process a single `jobs` remote channel. Jobs, identified by an id (`job_id`), are written to the channel. Each remotely executing task in this simulation reads a `job_id`, waits for a random amount of time and writes back a tuple of `job_id`, time taken and its own `pid` to the results channel. Finally all the `results` are printed out on the master process.

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));
```

```
julia> @everywhere function do_work(jobs, results) # define work function everywhere
           while true
               job_id = take!(jobs)
               exec_time = rand()
               sleep(exec_time) # simulates elapsed time doing actual work
               put!(results, (job_id, exec_time, myid()))
           end
       end

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> @async make_jobs(n); # feed the jobs channel with "n" jobs

julia> for p in workers() # start tasks on the workers to process requests in parallel
           remote_do(do_work, p, jobs, results)
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time, where = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds on worker $where")
           global n = n - 1
       end
1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
```

```
10 finished in 0.58 seconds on worker 2
0.055971741
```

Remote References and Distributed Garbage Collection

Objects referred to by remote references can be freed only when all held references in the cluster are deleted.

The node where the value is stored keeps track of which of the workers have a reference to it. Every time a `RemoteChannel` or a (unfetched) `Future` is serialized to a worker, the node pointed to by the reference is notified. And every time a `RemoteChannel` or a (unfetched) `Future` is garbage collected locally, the node owning the value is again notified. This is implemented in an internal cluster aware serializer. Remote references are only valid in the context of a running cluster. Serializing and deserializing references to and from regular `IO` objects is not supported.

The notifications are done via sending of "tracking" messages—an "add reference" message when a reference is serialized to a different process and a "delete reference" message when a reference is locally garbage collected.

Since `Future`s are write-once and cached locally, the act of `fetch`ing a `Future` also updates reference tracking information on the node owning the value.

The node which owns the value frees it once all references to it are cleared.

With `Future`s, serializing an already fetched `Future` to a different node also sends the value since the original remote store may have collected the value by this time.

It is important to note that when an object is locally garbage collected depends on the size of the object and the current memory pressure in the system.

In case of remote references, the size of the local reference object is quite small, while the value stored on the remote node may be quite large. Since the local object may not be collected immediately, it is a good practice to explicitly call `finalize` on local instances of a `RemoteChannel`, or on unfetched `Future`s. Since calling `fetch` on a `Future` also removes its reference from the remote store, this is not required on fetched `Future`s. Explicitly calling `finalize` results in an immediate message sent to the remote node to go ahead and remove its reference to the value.

Once finalized, a reference becomes invalid and cannot be used in any further calls.

25.8 Local invocations(@id man-distributed-local-invocations)

Data is necessarily copied over to the remote node for execution. This is the case for both remotecalls and when data is stored to a`RemoteChannel` / `Future` on a different node. As expected, this results in a copy of the serialized objects on the remote node. However, when the destination node is the local node, i.e. the calling process id is the same as the remote node id, it is executed as a local call. It is usually(not always) executed in a different task - but there is no

serialization/deserialization of data. Consequently, the call refers to the same object instances as passed - no copies are created. This behavior is highlighted below:

```
julia> using Distributed;

julia> rc = RemoteChannel(()->Channel(3));   # RemoteChannel created on local node

julia> v = [0];

julia> for i in 1:3
           v[1] = i                          # Reusing `v`
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[3], [3], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 1

julia> addprocs(1);

julia> rc = RemoteChannel(()->Channel(3), workers()[1]);   # RemoteChannel created on remote node

julia> v = [0];

julia> for i in 1:3
           v[1] = i
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[1], [2], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 3
```

As can be seen, `put!` on a locally owned `RemoteChannel` with the same object `v` modifed between calls results in the same single object instance stored. As opposed to copies of `v` being created when the node owning `rc` is a different node.

It is to be noted that this is generally not an issue. It is something to be factored in only if the object is both being stored locally and modifed post the call. In such cases it may be appropriate to store a `deepcopy` of the object.

This is also true for remotecalls on the local node as seen in the following example:

```
julia> using Distributed; addprocs(1);

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), myid(), v);     # Executed on local node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[1], v2=[1], true

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), workers()[1], v); # Executed on remote node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[0], v2=[1], false
```

As can be seen once again, a remote call onto the local node behaves just like a direct invocation. The call modifies local objects passed as arguments. In the remote invocation, it operates on a copy of the arguments.

To repeat, in general this is not an issue. If the local node is also being used as a compute node, and the arguments used post the call, this behavior needs to be factored in and if required deep copies of arguments must be passed to the call invoked on the local node. Calls on remote nodes will always operate on copies of arguments.

25.9 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a `DArray`, the behavior of a `SharedArray` is quite different. In a `DArray`, each process has local access to just a chunk of the data, and no two processes share the same chunk; in contrast, in a `SharedArray` each "participating" process has access to the entire array. A `SharedArray` is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

Shared Array support is available via module **SharedArrays** which must be explicitly loaded on all participating workers.

SharedArray indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on SharedArrays, albeit in single-process mode. In cases where an algorithm insists on an Array input, the underlying array can be retrieved from a SharedArray by calling sdata. For other **AbstractArray** types, sdata just returns the object itself, so it's safe to use sdata on any **Array**-type object.

The constructor for a shared array is of the form:

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

which creates an **N**-dimensional shared array of a bits type **T** and size **dims** across the processes specified by **pids**. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the **pids** named argument (and the creating process too, if it is on the same host).

If an **init** function, of signature **initfn(S::SharedArray)**, is specified, it is called on all the participating workers. You can specify that each worker runs the **init** function on a distinct portion of the array, thereby parallelizing initialization.

Here's a brief example:

```
julia> using Distributed

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S -> S[localindices(S)] = myid())
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
```

```
7


julia> S
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

`SharedArrays.localindices` provides disjoint one-dimensional ranges of indices, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

```
julia> S = SharedArray{Int,2}((3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] = myid())
3×4 SharedArray{Int64,2}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
@sync begin
    for p in procs(S)
        @async begin
            remotecall_wait(fill!, p, S, p)
        end
    end
end
```

would result in undefined behavior. Because each process fills the entire array with its own `pid`, whichever process is the last to execute (for any particular element of `S`) will have its `pid` retained.

As a more extended and complex example, consider running the following "kernel" in parallel:

```
q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if `q[i,j,t]` is near the end of the block assigned to one worker and `q[i,j,t+1]` is near the beginning of the block assigned to another, it's very likely that `q[i,j,t]` will not be ready at the time it's needed for computing `q[i,j,t+1]`. In such cases, one is better off chunking the array manually. Let's split along the second dimension. Define a function that returns the `(irange, jrange)` indices assigned to this worker:

```
julia> @everywhere function myrange(q::SharedArray)
           idx = indexpids(q)
           if idx == 0 # This worker is not assigned a piece
               return 1:0, 1:0
           end
           nchunks = length(procs(q))
           splits = [round(Int, s) for s in range(0, stop=size(q,2), length=nchunks+1)]
           1:size(q,1), splits[idx]+1:splits[idx+1]
       end
```

Next, define the kernel:

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange, trange)
           @show (irange, jrange, trange)  # display so we can see what's happening
           for t in trange, j in jrange, i in irange
               q[i,j,t+1] = q[i,j,t] + u[i,j,t]
           end
           q
       end
```

We also define a convenience wrapper for a `SharedArray` implementation

```
julia> @everywhere advection_shared_chunk!(q, u) =
           advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

Now let's compare three different versions, one that runs in a single process:

```
julia> advection_serial!(q, u) = advection_chunk!(q, u, 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

one that uses `@distributed`:

```
julia> function advection_parallel!(q, u)
           for t = 1:size(q,3)-1
               @sync @distributed for j = 1:size(q,2)
                   for i = 1:size(q,1)
                       q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                   end
               end
```

```
            end
            q
        end;
```

and one that delegates in chunks:

```
julia> function advection_shared!(q, u)
            @sync begin
                for p in procs(q)
                    @async remotecall_wait(advection_shared_chunk!, p, q, u)
                end
            end
            q
        end;
```

If we create `SharedArray`s and time these functions, we get the following results (with `julia -p 4`):

```
julia> q = SharedArray{Float64,3}((500,500,500));

julia> u = SharedArray{Float64,3}((500,500,500));
```

Run the functions once to JIT-compile and `@time` them on the second run:

```
julia> @time advection_serial!(q, u);
(irange,jrange,trange) = (1:500,1:500,1:499)
 830.220 milliseconds (216 allocations: 13820 bytes)

julia> @time advection_parallel!(q, u);
   2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)

julia> @time advection_shared!(q,u);
        From worker 2:       (irange,jrange,trange) = (1:500,1:125,1:499)
        From worker 4:       (irange,jrange,trange) = (1:500,251:375,1:499)
        From worker 3:       (irange,jrange,trange) = (1:500,126:250,1:499)
        From worker 5:       (irange,jrange,trange) = (1:500,376:500,1:499)
 238.119 milliseconds (2264 allocations: 169 KB)
```

The biggest advantage of `advection_shared!` is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

Shared Arrays and Distributed Garbage Collection

Like remote references, shared arrays are also dependent on garbage collection on the creating node to release references from all participating workers. Code which creates many short lived shared array objects would benefit from explicitly finalizing these objects as soon as possible. This results in both memory and file handles mapping the shared segment being released sooner.

25.10 ClusterManagers

The launching, management and networking of Julia processes into a logical cluster is done via cluster managers. A `ClusterManager` is responsible for

- launching worker processes in a cluster environment
- managing events during the lifetime of each worker
- optionally, providing data transport

A Julia cluster has the following characteristics:

- The initial Julia process, also called the `master`, is special and has an `id` of 1.
- Only the `master` process can add or remove worker processes.
- All processes can directly communicate with each other.

Connections between workers (using the in-built TCP/IP transport) is established in the following manner:

- `addprocs` is called on the master process with a `ClusterManager` object.
- `addprocs` calls the appropriate `launch` method which spawns required number of worker processes on appropriate machines.
- Each worker starts listening on a free port and writes out its host and port information to `stdout`.
- The cluster manager captures the `stdout` of each worker and makes it available to the master process.
- The master process parses this information and sets up TCP/IP connections to each worker.
- Every worker is also notified of other workers in the cluster.
- Each worker connects to all workers whose `id` is less than the worker's own `id`.

- In this way a mesh network is established, wherein every worker is directly connected with every other worker.

While the default transport layer uses plain `TCPSocket`, it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

- `LocalManager`, used when `addprocs()` or `addprocs(np::Integer)` are called
- `SSHManager`, used when `addprocs(hostnames::Array)` is called with a list of hostnames

`LocalManager` is used to launch additional workers on the same host, thereby leveraging multi-core and multi-processor hardware.

Thus, a minimal cluster manager would need to:

- be a subtype of the abstract `ClusterManager`
- implement `launch`, a method responsible for launching new workers
- implement `manage`, which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

`addprocs(manager::FooManager)` requires `FooManager` to implement:

```
function launch(manager::FooManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

As an example let us see how the `LocalManager`, the manager responsible for starting workers on the same host, is implemented:

```
struct LocalManager <: ClusterManager
    np::Integer
end

function launch(manager::LocalManager, params::Dict, launched::Array, c::Condition)
```

```
    [...]
end

function manage(manager::LocalManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

The launch method takes the following arguments:

- `manager::ClusterManager`: the cluster manager that addprocs is called with
- `params::Dict`: all the keyword arguments passed to addprocs
- `launched::Array`: the array to append one or more `WorkerConfig` objects to
- `c::Condition`: the condition variable to be notified as and when workers are launched

The launch method is called asynchronously in a separate task. The termination of this task signals that all requested workers have been launched. Hence the launch function MUST exit as soon as all the requested workers have been launched.

Newly launched workers are connected to each other and the master process in an all-to-all manner. Specifying the command line argument `--worker[=<cookie>]` results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets.

All workers in a cluster share the same cookie as the master. When the cookie is unspecified, i.e, with the `--worker` option, the worker tries to read it from its standard input. `LocalManager` and `SSHManager` both pass the cookie to newly launched workers via their standard inputs.

By default a worker will listen on a free port at the address returned by a call to `getipaddr()`. A specific address to listen on may be specified by optional argument `--bind-to bind_addr[:port]`. This is useful for multi-homed hosts.

As an example of a non-TCP/IP transport, an implementation may choose to use MPI, in which case `--worker` must NOT be specified. Instead, newly launched workers should call `init_worker(cookie)` before using any of the parallel constructs.

For every worker launched, the launch method must add a `WorkerConfig` object (with appropriate fields initialized) to `launched`

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
```

```
    io::Union{IO, Nothing}
    host::Union{AbstractString, Nothing}
    port::Union{Integer, Nothing}

    # Used when launching additional workers at a host
    count::Union{Int, Symbol, Nothing}
    exename::Union{AbstractString, Cmd, Nothing}
    exeflags::Union{Cmd, Nothing}

    # External cluster managers can use this to store information at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Any

    # SSHManager / SSH tunnel connections to workers
    tunnel::Union{Bool, Nothing}
    bind_addr::Union{AbstractString, Nothing}
    sshflags::Union{Cmd, Nothing}
    max_parallel::Union{Integer, Nothing}

    # Used by Local/SSH managers
    connect_at::Any

    [...]
end
```

Most of the fields in `WorkerConfig` are used by the inbuilt managers. Custom cluster managers would typically specify only `io` or `host` / `port`:

- If `io` is specified, it is used to read host/port information. A Julia worker prints out its bind address and port at startup. This allows Julia workers to listen on any free port available instead of requiring worker ports to be configured manually.
- If `io` is not specified, `host` and `port` are used to connect.
- `count`, `exename` and `exeflags` are relevant for launching additional workers from a worker. For example, a cluster manager may launch a single worker per node, and use that to launch additional workers.
  - `count` with an integer value `n` will launch a total of `n` workers.
  - `count` with a value of `:auto` will launch as many workers as the number of CPU threads (logical cores) on that machine.

  - `exename` is the name of the `julia` executable including the full path.
  - `exeflags` should be set to the required command line arguments for new workers.

- `tunnel`, `bind_addr`, `sshflags` and `max_parallel` are used when a ssh tunnel is required to connect to the workers from the master process.
- `userdata` is provided for custom cluster managers to store their own worker-specific information.

`manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)` is called at different times during the worker's lifetime with appropriate `op` values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

Cluster Managers with Custom Transports

Replacing the default TCP/IP all-to-all socket connections with a custom transport layer is a little more involved. Each Julia process has as many communication tasks as the workers it is connected to. For example, consider a Julia cluster of 32 processes in an all-to-all mesh network:

- Each Julia process thus has 31 communication tasks.
- Each task handles all incoming messages from a single remote worker in a message-processing loop.
- The message-processing loop waits on an `IO` object (for example, a `TCPSocket` in the default implementation), reads an entire message, processes it and waits for the next one.
- Sending messages to a process is done directly from any Julia task—not just communication tasks—again, via the appropriate `IO` object.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate `IO` objects that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

The default implementation (which uses TCP/IP sockets) is implemented as `connect(manager::ClusterManager, pid::Integer, config::WorkerConfig)`.

`connect` should return a pair of `IO` objects, one for reading data sent from worker `pid`, and the other to write data that needs to be sent to worker `pid`. Custom cluster managers can use an in-memory `BufferStream` as the plumbing to proxy data between the custom, possibly non-`IO` transport and Julia's in-built parallel infrastructure.

A `BufferStream` is an in-memory `IOBuffer` which behaves like an `IO`–it is a stream which can be handled asynchronously.

The folder `clustermanager/0mq` in the Examples repository contains an example of using ZeroMQ to connect Julia workers in a star topology with a 0MQ broker in the middle. Note: The Julia processes are still all logically connected to each other–any worker can message any other worker directly without any awareness of 0MQ being used as the transport layer.

When using custom transports:

- Julia workers must NOT be started with `--worker`. Starting with `--worker` will result in the newly launched workers defaulting to the TCP/IP socket transport implementation.
- For every incoming logical connection with a worker, `Base.process_messages(rd::IO, wr::IO)()` must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the `IO` objects.
- `init_worker(cookie, manager::FooManager)` must be called as part of worker process initialization.
- Field `connect_at::Any` in `WorkerConfig` can be set by the cluster manager when `launch` is called. The value of this field is passed in in all `connect` callbacks. Typically, it carries information on how to connect to a worker. For example, the TCP/IP socket transport uses this field to specify the `(host, port)` tuple at which to connect to a worker.

`kill(manager, pid, config)` is called to remove a worker from the cluster. On the master process, the corresponding `IO` objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an `exit()` call on the specified remote worker.

The Examples folder `clustermanager/simple` is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

Network Requirements for LocalManager and SSHManager

Julia clusters are designed to be executed on already secured environments on infrastructure such as local laptops, departmental clusters, or even the cloud. This section covers network security requirements for the inbuilt `LocalManager` and `SSHManager`:

- The master process does not listen on any port. It only connects out to the workers.
- Each worker binds to only one of the local interfaces and listens on an ephemeral port number assigned by the OS.
- `LocalManager`, used by `addprocs(N)`, by default binds only to the loopback interface. This means that workers started later on remote hosts (or by anyone with malicious intentions) are unable to connect to the cluster. An `addprocs(4)` followed by an `addprocs(["remote_host"])` will fail. Some users may need to create a cluster comprising their local system and a few remote systems. This can be done by explicitly requesting `LocalManager` to bind to an external network interface via the `restrict` keyword argument: `addprocs(4; restrict=false)`.
- `SSHManager`, used by `addprocs(list_of_remote_hosts)`, launches workers on remote hosts via SSH. By default SSH is only used to launch Julia workers. Subsequent master-worker and worker-worker connections use plain, unencrypted TCP/IP sockets. The remote hosts must have passwordless login enabled. Additional SSH flags or credentials may be specified via keyword argument `sshflags`.
- `addprocs(list_of_remote_hosts; tunnel=true, sshflags=<ssh keys and other flags>)` is useful when we wish to use SSH connections for master-worker too. A typical scenario for this is a local laptop running the Julia REPL (i.e., the master) with the rest of the cluster on the cloud, say on Amazon EC2. In this case only port 22 needs to be opened at the remote cluster coupled with SSH client authenticated via public key infrastructure (PKI). Authentication credentials can be supplied via `sshflags`, for example `sshflags=`-i <keyfile>``.

  In an all-to-all topology (the default), all workers connect to each other via plain TCP sockets. The security policy on the cluster nodes must thus ensure free connectivity between workers for the ephemeral port range (varies by OS).

  Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom `ClusterManager`.

Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

- `cluster_cookie()` returns the cookie, while `cluster_cookie(cookie)()` sets it and returns the new cookie.
- All connections are authenticated on both sides to ensure that only workers started by the master are allowed to connect to each other.
- The cookie may be passed to the workers at startup via argument `--worker=<cookie>`. If argument `--worker` is specified without the cookie, the worker tries to read the cookie from its standard input (`stdin`). The `stdin` is closed immediately after the cookie is retrieved.

- `ClusterManager`s can retrieve the cookie on the master by calling `cluster_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `--worker`) must call `init_worker(cookie, manager)` with the same cookie as on the master.

Note that environments requiring higher levels of security can implement this via a custom `ClusterManager`. For example, cookies can be pre-shared and hence not specified as a startup argument.

25.11 Specifying Network Topology (Experimental)

The keyword argument `topology` passed to `addprocs` is used to specify how the workers must be connected to each other:

- `:all_to_all`, the default: all workers are connected to each other.
- `:master_worker`: only the driver process, i.e. `pid` 1, has connections to the workers.
- `:custom`: the `launch` method of the cluster manager specifies the connection topology via the fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster-manager-provided identity `ident` will connect to all workers specified in `connect_idents`.

Keyword argument `lazy=true|false` only affects `topology` option `:all_to_all`. If `true`, the cluster starts off with the master connected to all workers. Specific worker-worker connections are established at the first remote invocation between two workers. This helps in reducing initial resources allocated for intra-cluster communication. Connections are setup depending on the runtime requirements of a parallel program. Default value for `lazy` is `true`.

Currently, sending a message between unconnected workers results in an error. This behaviour, as with the functionality and interface, should be considered experimental in nature and may change in future releases.

25.12 Noteworthy external packages

Outside of Julia parallelism there are plenty of external packages that should be mentioned. For example MPI.jl is a Julia wrapper for the `MPI` protocol, or DistributedArrays.jl, as presented in Shared Arrays. A mention must be made of Julia's GPU programming ecosystem, which includes:

1. Low-level (C kernel) based operations OpenCL.jl and CUDAdrv.jl which are respectively an OpenCL interface and a CUDA wrapper.
2. Low-level (Julia Kernel) interfaces like CUDAnative.jl which is a Julia native CUDA implementation.
3. High-level vendor-specific abstractions like CuArrays.jl and CLArrays.jl

4. High-level libraries like ArrayFire.jl and GPUArrays.jl

In the following example we will use both `DistributedArrays.jl` and `CuArrays.jl` to distribute an array across multiple processes by first casting it through `distribute()` and `CuArray()`.

Remember when importing `DistributedArrays.jl` to import it across all processes using `@everywhere`

```
$ ./julia -p 4

julia> addprocs()

julia> @everywhere using DistributedArrays

julia> using CuArrays

julia> B = ones(10_000) ./ 2;

julia> A = ones(10_000) .* π;

julia> C = 2 .* A ./ B;

julia> all(C .≈ 4*π)
true

julia> typeof(C)
Array{Float64,1}

julia> dB = distribute(B);

julia> dA = distribute(A);

julia> dC = 2 .* dA ./ dB;

julia> all(dC .≈ 4*π)
true

julia> typeof(dC)
DistributedArrays.DArray{Float64,1,Array{Float64,1}}

julia> cuB = CuArray(B);
```

```
julia> cuA = CuArray(A);

julia> cuC = 2 .* cuA ./ cuB;

julia> all(cuC .≈ 4*π);
true

julia> typeof(cuC)
CuArray{Float64,1}
```

Keep in mind that some Julia features are not currently supported by CUDAnative.jl [2] , especially some functions like `sin` will need to be replaced with `CUDAnative.sin`(cc: @maleadt).

In the following example we will use both `DistributedArrays.jl` and `CuArrays.jl` to distribute an array across multiple processes and call a generic function on it.

```
function power_method(M, v)
    for i in 1:100
        v = M*v
        v /= norm(v)
    end

    return v, norm(M*v) / norm(v)  # or  (M*v) ./ v
end
```

`power_method` repeatedly creates a new vector and normalizes it. We have not specified any type signature in function declaration, let's see if it works with the aforementioned datatypes:

```
julia> M = [2. 1; 1 1];

julia> v = rand(2)
2-element Array{Float64,1}:
0.40395
0.445877

julia> power_method(M,v)
([0.850651, 0.525731], 2.618033988749895)
```

```
julia> cuM = CuArray(M);

julia> cuv = CuArray(v);

julia> curesult = power_method(cuM, cuv);

julia> typeof(curesult)
CuArray{Float64,1}

julia> dM = distribute(M);

julia> dv = distribute(v);

julia> dC = power_method(dM, dv);

julia> typeof(dC)
Tuple{DistributedArrays.DArray{Float64,1,Array{Float64,1}},Float64}
```

To end this short exposure to external packages, we can consider **`MPI.jl`**, a Julia wrapper of the MPI protocol. As it would take too long to consider every inner function, it would be better to simply appreciate the approach used to implement the protocol.

Consider this toy script which simply calls each subprocess, instantiate its rank and when the master process is reached, performs the ranks' sum

```
import MPI

MPI.Init()

comm = MPI.COMM_WORLD
MPI.Barrier(comm)

root = 0
r = MPI.Comm_rank(comm)

sr = MPI.Reduce(r, MPI.SUM, root, comm)

if(MPI.Comm_rank(comm) == root)
   @printf("sum of ranks: %s\n", sr)
end
```

```
MPI.Finalize()
```

```
mpirun -np 4 ./julia example.jl
```

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding rma to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see https://mpi-forum.org/docs.

[2] Julia GPU man pages

Chapter 26

Asynchronous Programming

When a program needs to interact with the outside world, for example communicating with another machine over the internet, operations in the program may need to happen in an unpredictable order. Say your program needs to download a file. We would like to initiate the download operation, perform other operations while we wait for it to complete, and then resume the code that needs the downloaded file when it is available. This sort of scenario falls in the domain of asynchronous programming, sometimes also referred to as concurrent programming (since, conceptually, multiple things are happening at once).

To address these scenarios, Julia provides Tasks (also known by several other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations). When a piece of computing work (in practice, executing a particular function) is designated as a Task, it becomes possible to interrupt it by switching to another Task. The original Task can later be resumed, at which point it will pick up right where it left off. At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

26.1 Basic `Task` operations

You can think of a `Task` as a handle to a unit of computational work to be performed. It has a create-start-run-finish lifecycle. Tasks are created by calling the `Task` constructor on a 0-argument function to run, or using the `@task` macro:

```
julia> t = @task begin; sleep(5); println("done"); end
Task (runnable) @0x00007f13a40c0eb0
```

`@task x` is equivalent to `Task(()->x)`.

This task will wait for five seconds, and then print `done`. However, it has not started running yet. We can run it whenever we're ready by calling `schedule`:

```
julia> schedule(t);
```

If you try this in the REPL, you will see that `schedule` returns immediately. That is because it simply adds `t` to an internal queue of tasks to run. Then, the REPL will print the next prompt and wait for more input. Waiting for keyboard input provides an opportunity for other tasks to run, so at that point `t` will start. `t` calls `sleep`, which sets a timer and stops execution. If other tasks have been scheduled, they could run then. After five seconds, the timer fires and restarts `t`, and you will see `done` printed. `t` is then finished.

The `wait` function blocks the calling task until some other task finishes. So for example if you type

```
julia> schedule(t); wait(t)
```

instead of only calling `schedule`, you will see a five second pause before the next input prompt appears. That is because the REPL is waiting for `t` to finish before proceeding.

It is common to want to create a task and schedule it right away, so the macro `@async` is provided for that purpose -- `@async x` is equivalent to `schedule(@task x)`.

26.2 Communicating with Channels

In some problems, the various pieces of required work are not naturally related by function calls; there is no obvious "caller" or "callee" among the jobs that need to be done. An example is the producer-consumer problem, where one complex procedure is generating values and another complex procedure is consuming them. The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return. With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.

Julia provides a `Channel` mechanism for solving this problem. A `Channel` is a waitable first-in first-out queue which can have multiple tasks reading from and writing to it.

Let's define a producer task, which produces values via the `put!` call. To consume values, we need to schedule the producer to run in a new task. A special `Channel` constructor which accepts a 1-arg function as an argument can be used to run a task bound to a channel. We can then `take!` values repeatedly from the channel object:

```
julia> function producer(c::Channel)
           put!(c, "start")
```

```
            for n=1:4
                put!(c, 2n)
            end
            put!(c, "stop")
        end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"

julia> take!(chnl)
2

julia> take!(chnl)
4

julia> take!(chnl)
6

julia> take!(chnl)
8

julia> take!(chnl)
"stop"
```

One way to think of this behavior is that `producer` was able to return multiple times. Between calls to `put!`, the producer's execution is suspended and the consumer has control.

The returned `Channel` can be used as an iterable object in a `for` loop, in which case the loop variable takes on all the produced values. The loop is terminated when the channel is closed.

```
julia> for x in Channel(producer)
            println(x)
        end
start
2
4
6
```

```
8
stop
```

Note that we did not have to explicitly close the channel in the producer. This is because the act of binding a `Channel` to a `Task` associates the open lifetime of a channel with that of the bound task. The channel object is closed automatically when the task terminates. Multiple channels can be bound to a task, and vice-versa.

While the `Task` constructor expects a 0-argument function, the `Channel` method that creates a task-bound channel expects a function that accepts a single argument of type `Channel`. A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0 or 1 argument anonymous function.

For `Task` objects this can be done either directly or by use of a convenience macro:

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

To orchestrate more advanced work distribution patterns, `bind` and `schedule` can be used in conjunction with `Task` and `Channel` constructors to explicitly link a set of channels with a set of producer/consumer tasks.

More on Channels

A channel can be visualized as a pipe, i.e., it has a write end and a read end :

- Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.
- Multiple readers in different tasks can read data concurrently via `take!` calls.
- As an example:

```
# Given Channels c1 and c2,
c1 = Channel(32)
c2 = Channel(32)

# and a function `foo` which reads items from c1, processes the item read
# and writes a result to c2,
function foo()
```

```
    while true
        data = take!(c1)
        [...]               # process data
        put!(c2, result)    # write out result
    end
end

# we can schedule `n` instances of `foo` to be active concurrently.
for _ in 1:n
    @async foo()
end
```

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of type `T`. If the type is not specified, the channel can hold objects of any type. `sz` refers to the maximum number of elements that can be held in the channel at any time. For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type. A `Channel{MyType}(64)` can hold up to 64 objects of `MyType` at any time.
- If a `Channel` is empty, readers (on a `take!` call) will block until data is available.
- If a `Channel` is full, writers (on a `put!` call) will block until space becomes available.
- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.
- A `Channel` is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls. `close` closes a `Channel`. On a closed `Channel`, `put!` will fail. For example:

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);

julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException("Channel is closed.",:closed)
Stacktrace:
[...]
```

- `take!` and `fetch` (which retrieves but does not remove the value) on a closed channel successfully return any existing values until it is emptied. Continuing the above example:

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1
```

```
julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
ERROR: InvalidStateException("Channel is closed.",:closed)
Stacktrace:
[...]
```

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single `jobs` channel. Jobs, identified by an id (`job_id`), are written to the channel. Each task in this simulation reads a `job_id`, waits for a random amount of time and writes back a tuple of `job_id` and the simulated time to the results channel. Finally all the `results` are printed out.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()
           for job_id in jobs
               exec_time = rand()
               sleep(exec_time)                # simulates elapsed time doing actual work
                                               # typically performed externally.
               put!(results, (job_id, exec_time))
           end
       end;

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> @async make_jobs(n); # feed the jobs channel with "n" jobs
```

```
julia> for i in 1:4 # start 4 tasks to process requests in parallel
           @async do_work()
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds")
           global n = n - 1
       end
4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
9 finished in 0.36 seconds
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

26.3 More task operations

Task operations are built on a low-level primitive called `yieldto`. `yieldto(task, value)` suspends the current task, switches to the specified `task`, and causes that task's last `yieldto` call to return the specified `value`. Notice that `yieldto` is the only operation required to use task-style control flow; instead of calling and returning we are always just switching to a different task. This is why this feature is also called "symmetric coroutines"; each task is switched to and from using the same mechanism.

`yieldto` is powerful, but most uses of tasks do not invoke it directly. Consider why this might be. If you switch away from the current task, you will probably want to switch back to it at some point, but knowing when to switch back, and knowing which task has the responsibility of switching back, can require considerable coordination. For example, `put!` and `take!` are blocking operations, which, when used in the context of channels maintain state to remember who the consumers are. Not needing to manually keep track of the consuming task is what makes `put!` easier to use than the low-level `yieldto`.

In addition to `yieldto`, a few other basic functions are needed to use tasks effectively.

- `current_task` gets a reference to the currently-running task.
- `istaskdone` queries whether a task has exited.
- `istaskstarted` queries whether a task has run yet.
- `task_local_storage` manipulates a key-value store specific to the current task.

26.4 Tasks and events

Most task switches occur as a result of waiting for events such as I/O requests, and are performed by a scheduler included in Julia Base. The scheduler maintains a queue of runnable tasks, and executes an event loop that restarts tasks based on external events such as message arrival.

The basic function for waiting for an event is `wait`. Several objects implement `wait`; for example, given a `Process` object, `wait` will wait for it to exit. `wait` is often implicit; for example, a `wait` can happen inside a call to `read` to wait for data to be available.

In all of these cases, `wait` ultimately operates on a `Condition` object, which is in charge of queueing and restarting tasks. When a task calls `wait` on a `Condition`, the task is marked as non-runnable, added to the condition's queue, and switches to the scheduler. The scheduler will then pick another task to run, or block waiting for external events. If all goes well, eventually an event handler will call `notify` on the condition, which causes tasks waiting for that condition to become runnable again.

A task created explicitly by calling `Task` is initially not known to the scheduler. This allows you to manage tasks manually using `yieldto` if you wish. However, when such a task waits for an event, it still gets restarted automatically when the event happens, as you would expect.

Chapter 27

Multi-Threading

Visit this blog post for a presentation of Julia multi-threading features.

27.1 Starting Julia with multiple threads

By default, Julia starts up with a single thread of execution. This can be verified by using the command `Threads.nthreads()`:

```
julia> Threads.nthreads()
1
```

The number of execution threads is controlled either by using the `-t`/`--threads` command line argument or by using the `JULIA_NUM_THREADS` environment variable. When both are specified, then `-t`/`--threads` takes precedence.

> Julia 1.5
>
> The `-t`/`--threads` command line argument requires at least Julia 1.5. In older versions you must use the environment variable instead.

Lets start Julia with 4 threads:

```
$ julia --threads 4
```

Let's verify there are 4 threads at our disposal.

```
julia> Threads.nthreads()
4
```

But we are currently on the master thread. To check, we use the function `Threads.threadid`

```
julia> Threads.threadid()
1
```

Note

If you prefer to use the environment variable you can set it as follows in Bash (Linux/macOS):

```
export JULIA_NUM_THREADS=4
```

C shell on Linux/macOS, CMD on Windows:

```
set JULIA_NUM_THREADS=4
```

Powershell on Windows:

```
$env:JULIA_NUM_THREADS=4
```

Note that this must be done before starting Julia.

Note

The number of threads specified with `-t`/`--threads` is propagated to worker processes that are spawned using the `-p`/`--procs` or `--machine-file` command line options. For example, `julia -p2 -t2` spawns 1 main process with 2 worker processes, and all three processes have 2 threads enabled. For more fine grained control over worker threads use `addprocs` and pass `-t`/`--threads` as `exeflags`.

27.2 Data-race freedom

You are entirely responsible for ensuring that your program is data-race free, and nothing promised here can be assumed if you do not observe that requirement. The observed results may be highly unintuitive.

The best way to ensure this is to acquire a lock around any access to data that can be observed from multiple threads. For example, in most cases you should use the following code pattern:

```
julia> lock(lk) do
           use(a)
       end

julia> begin
           lock(lk)
           try
               use(a)
           finally
```

```
            unlock(lk)
        end
    end
```

where `lk` is a lock (e.g. `ReentrantLock()`) and `a` data.

Additionally, Julia is not memory safe in the presence of a data race. Be very careful about reading any data if another thread might write to it! Instead, always use the lock pattern above when changing data (such as assigning to a global or closure variable) accessed by other threads.

```
Thread 1:
global b = false
global a = rand()
global b = true

Thread 2:
while !b; end
bad_read1(a) # it is NOT safe to access `a` here!

Thread 3:
while !@isdefined(a); end
bad_read2(a) # it is NOT safe to access `a` here
```

27.3 The `@threads` Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
julia> a = zeros(10)
10-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the `Threads.@threads` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
julia> Threads.@threads for i = 1:10
           a[i] = Threads.threadid()
       end
```

The iteration space is split among the threads, after which each thread writes its thread ID to its assigned locations:

```
julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Note that `Threads.@threads` does not have an optional reduction parameter like `@distributed`.

27.4 Atomic Operations

Julia supports accessing and modifying values atomically, that is, in a thread-safe way to avoid race conditions. A value (which must be of a primitive type) can be wrapped as `Threads.Atomic` to indicate it must be accessed in this way. Here we can see an example:

```
julia> i = Threads.Atomic{Int}(0);

julia> ids = zeros(4);

julia> old_is = zeros(4);
```

```
julia> Threads.@threads for id in 1:4
           old_is[id] = Threads.atomic_add!(i, id)
           ids[id] = id
       end

julia> old_is
4-element Array{Float64,1}:
 0.0
 1.0
 7.0
 3.0

julia> ids
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
```

Had we tried to do the addition without the atomic tag, we might have gotten the wrong answer due to a race condition. An example of what would happen if we didn't avoid the race:

```
julia> using Base.Threads

julia> nthreads()
4

julia> acc = Ref(0)
Base.RefValue{Int64}(0)

julia> @threads for i in 1:1000
          acc[] += 1
       end

julia> acc[]
926

julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)
```

```
julia> @threads for i in 1:1000
          atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

> Note
>
> Not all primitive types can be wrapped in an `Atomic` tag. Supported types are `Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `Float16`, `Float32`, and `Float64`. Additionally, `Int128` and `UInt128` are not supported on AAarch32 and ppc64le.

27.5 Side effects and mutable function arguments

When using multi-threading we have to be careful when using functions that are not pure as we might get a wrong answer. For instance functions that have a name ending with ! by convention modify their arguments and thus are not pure.

27.6 @threadcall

External libraries, such as those called via `ccall`, pose a problem for Julia's task-based I/O mechanism. If a C library performs a blocking operation, that prevents the Julia scheduler from executing any other tasks until the call returns. (Exceptions are calls into custom C code that call back into Julia, which may then yield, or C code that calls `jl_yield()`, the C equivalent of `yield`.)

The `@threadcall` macro provides a way to avoid stalling execution in such a scenario. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable `UV_THREADPOOL_SIZE`. While waiting for a free thread, and during function execution once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that `@threadcall` does not return until the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia, as it will segfault.

`@threadcall` may be removed/changed in future versions of Julia.

27.7 Caveats

At this time, most operations in the Julia runtime and standard libraries can be used in a thread-safe manner, if the user code is data-race free. However, in some areas work on stabilizing thread support is ongoing. Multi-threaded

programming has many inherent difficulties, and if a program using threads exhibits unusual or undesirable behavior (e.g. crashes or mysterious results), thread interactions should typically be suspected first.

There are a few specific limitations and warnings to be aware of when using threads in Julia:

- Base collection types require manual locking if used simultaneously by multiple threads where at least one thread modifies the collection (common examples include `push!` on arrays, or inserting items into a `Dict`).
- After a task starts running on a certain thread (e.g. via `@spawn`), it will always be restarted on the same thread after blocking. In the future this limitation will be removed, and tasks will migrate between threads.
- `@threads` currently uses a static schedule, using all threads and assigning equal iteration counts to each. In the future the default schedule is likely to change to be dynamic.
- The schedule used by `@spawn` is nondeterministic and should not be relied on.
- Compute-bound, non-memory-allocating tasks can prevent garbage collection from running in other threads that are allocating memory. In these cases it may be necessary to insert a manual call to `GC.safepoint()` to allow GC to run. This limitation will be removed in the future.
- Avoid running top-level operations, e.g. `include`, or `eval` of type, method, and module definitions in parallel.
- Be aware that finalizers registered by a library may break if threads are enabled. This may require some transitional work across the ecosystem before threading can be widely adopted with confidence. See the next section for further details.

27.8 Safe use of Finalizers

Because finalizers can interrupt any code, they must be very careful in how they interact with any global state. Unfortunately, the main reason that finalizers are used is to update global state (a pure function is generally rather pointless as a finalizer). This leads us to a bit of a conundrum. There are a few approaches to dealing with this problem:

1. When single-threaded, code could call the internal `jl_gc_enable_finalizers` C function to prevent finalizers from being scheduled inside a critical region. Internally, this is used inside some functions (such as our C locks) to prevent recursion when doing certain operations (incremental package loading, codegen, etc.). The combination of a lock and this flag can be used to make finalizers safe.
2. A second strategy, employed by Base in a couple places, is to explicitly delay a finalizer until it may be able to acquire its lock non-recursively. The following example demonstrates how this strategy could be applied to `Distributed.finalize_ref`:

```
function finalize_ref(r::AbstractRemoteRef)
    if r.where > 0 # Check if the finalizer is already run
        if islocked(client_refs) || !trylock(client_refs)
            # delay finalizer for later if we aren't free to acquire the lock
            finalizer(finalize_ref, r)
            return nothing
        end
        try # `lock` should always be followed by `try`
            if r.where > 0 # Must check again here
                # Do actual cleanup here
                r.where = 0
            end
        finally
            unlock(client_refs)
        end
    end
    nothing
end
```

3. A related third strategy is to use a yield-free queue. We don't currently have a lock-free queue implemented in Base, but `Base.InvasiveLinkedListSynchronized{T}` is suitable. This can frequently be a good strategy to use for code with event loops. For example, this strategy is employed by `Gtk.jl` to manage lifetime ref-counting. In this approach, we don't do any explicit work inside the `finalizer`, and instead add it to a queue to run at a safer time. In fact, Julia's task scheduler already uses this, so defining the finalizer as `x -> @spawn do_cleanup(x)` is one example of this approach. Note however that this doesn't control which thread `do_cleanup` runs on, so `do_cleanup` would still need to acquire a lock. That doesn't need to be true if you implement your own queue, as you can explicitly only drain that queue from your thread.

Chapter 28

Multi-processing and Distributed Computing

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI[1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Distributed programming in Julia is built on two primitives: remote references and remote calls. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: `Future` and `RemoteChannel`.

A remote call returns a `Future` to its result. Remote calls return immediately; the process that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling `wait` on the returned `Future`, and you can obtain the full value of the result using `fetch`.

On the other hand, `RemoteChannel` s are rewritable. For example, multiple processes can co-ordinate their processing by referencing the same remote `Channel`.

Each process has an associated identifier. The process providing the interactive Julia prompt always has an `id` equal to 1. The processes used by default for parallel operations are referred to as "workers". When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1. As a result, adding 2 or more processes is required to gain benefits from parallel processing methods like `pmap`. Adding a single process is beneficial if you just wish to do other things in the main process while a long computation is running on the worker.

Let's try this out. Starting with `julia -p n` provides `n` worker processes on the local machine. Generally it makes sense for `n` to equal the number of CPU threads (logical cores) on the machine. Note that the `-p` argument implicitly loads module `Distributed`.

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, nothing)

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

The first argument to `remotecall` is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but `remotecall` is considered a low-level interface providing finer control. The second argument to `remotecall` is the `id` of the process that will do the work, and the remaining arguments will be passed to the function being called.

As you can see, in the first line we asked process 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two futures, `r` and `s`. The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function `remotecall_fetch` exists for this purpose. It is equivalent to `fetch(remotecall(...))` but is more efficient.

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.18526337335308085
```

Remember that `getindex(r,1,1)` is equivalent to `r[1,1]`, so this call fetches the first element of the future `r`.

To make things easier, the symbol `:any` can be passed to `@spawnat`, which picks where to do the operation for you:

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
Future(3, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

Note that we used `1 .+ fetch(r)` instead of `1 .+ r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the process doing the addition. In this case, `@spawnat` is smart enough to perform the computation on the process that owns `r`, so the `fetch` will be a no-op (no work is done).

(It is worth noting that `@spawnat` is not built-in but defined in Julia as a macro. It is possible to define your own such constructs.)

An important thing to remember is that, once fetched, a `Future` will cache its value locally. Further `fetch` calls do not entail a network hop. Once all referencing `Future`s have fetched, the remote stored value is deleted.

`@async` is similar to `@spawnat`, but only runs tasks on the local process. We use it to create a "feeder" task for each process. Each task picks the next index that needs to be computed, then waits for its process to finish, then repeats until we run out of indices. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `@sync` block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. As for v0.7 and beyond, the feeder tasks are able to share state via `nextidx` because they all run on the same process. Even if `Tasks` are scheduled cooperatively, locking may still be required in some contexts, as in asynchronous I/O. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch` is called. This is the current state of implementation and it may change for future Julia versions, as it is intended to make it possible to run up to N `Tasks` on M `Process`, aka M:N Threading. Then a lock acquiring‒releasing model for `nextidx` will be needed, as it is not safe to let multiple processes read-write a resource at the same time.

28.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

```
julia> function rand2(dims...)
           return 2*rand(dims...)
       end

julia> rand2(2,2)
2×2 Array{Float64,2}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawnat :any rand2(2,2))
ERROR: RemoteException(2, CapturedException(UndefVarError(Symbol("#rand2"))
Stacktrace:
[...]
```

Process 1 knew about the function `rand2`, but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, `DummyModule.jl`, containing the following code:

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

In order to refer to `MyType` across all processes, `DummyModule.jl` needs to be loaded on every process. Calling `include("DummyModule.jl")` loads it only on a single process. To load it on every process, use the `@everywhere` macro (starting Julia with `julia -p 2`):

```
julia> @everywhere include("DummyModule.jl")
loaded
      From worker 3:    loaded
      From worker 2:    loaded
```

As usual, this does not bring `DummyModule` into scope on any of the process, which requires `using` or `import`. Moreover, when `DummyModule` is brought into scope on one process, it is not on any other:

```
julia> using .DummyModule

julia> MyType(7)
MyType(7)

julia> fetch(@spawnat 2 MyType(7))
ERROR: On worker 2:
UndefVarError: MyType not defined
⋮

julia> fetch(@spawnat 2 DummyModule.MyType(7))
MyType(7)
```

However, it's still possible, for instance, to send a `MyType` to a process which has loaded `DummyModule` even if it's not in scope:

```
julia> put!(RemoteChannel(2), MyType(7))
RemoteChannel{Channel{Any}}(2, 1, 13)
```

A file can also be preloaded on multiple processes at startup with the `-L` flag, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

The Julia process running the driver script in the example above has an `id` equal to 1, just like a process providing an interactive prompt.

Finally, if `DummyModule.jl` is not a standalone file but a package, then `using DummyModule` will load `DummyModule.jl` on all processes, but only bring it into scope on the process where `using` was called.

28.2 Starting and managing worker processes

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the `-p` option as shown above.
- A cluster spanning machines using the `--machine-file` option. This uses a passwordless `ssh` login to start Julia worker processes (from the same path as the current host) on the specified machines. Each machine definition takes the form `[count*][user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. `count` is the number of workers to spawn on the node, and defaults to 1. The optional `bind-to bind_addr[:port]` specifies the IP address and port that other workers should use to connect to this worker.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

```
julia> using Distributed

julia> addprocs(2)
2-element Array{Int64,1}:
 2
 3
```

Module `Distributed` must be explicitly loaded on the master process before invoking `addprocs`. It is automatically made available on the worker processes.

Note that workers do not run a `~/.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes. You may use `addprocs(exeflags="--project")` to initialize a worker with a particular environment, and then `@everywhere using <modulename>` or `@everywhere include("file.jl")`.

Other types of clusters can be supported by writing your own custom `ClusterManager`, as described below in the ClusterManagers section.

28.3 Data Movement

Sending messages and moving data constitute most of the overhead in a distributed program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various distributed programming constructs.

`fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `@spawnat` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

Method 1:

```
julia> A = rand(1000,1000);

julia> Bref = @spawnat :any A^2;

[...]

julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawnat :any rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawnat`. In the first method, a random matrix is constructed locally, then sent to another process where it is squared. In the second method, a random matrix is both constructed and squared on another process. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and likely some measurement. For example, if the first process needs matrix `A` then the first method might be better. Or, if computing `A` is expensive and only the current process has it, then moving it to another process might be unavoidable. Or, if the current process has very little to do between the `@spawnat` and `fetch(Bref)`, it might be better to eliminate the parallelism altogether. Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawnat` statement just for this step.

28.4 Global variables

Expressions executed remotely via `@spawnat`, or closures specified for remote execution using `remotecall` may refer to global variables. Global bindings under module `Main` are treated a little differently compared to global bindings in other modules. Consider the following code snippet:

```
A = rand(10,10)
remotecall_fetch(()->sum(A), 2)
```

In this case `sum` MUST be defined in the remote process. Note that `A` is a global variable defined in the local workspace. Worker 2 does not have a variable called `A` under `Main`. The act of shipping the closure `()->sum(A)` to worker 2 results in `Main.A` being defined on 2. `Main.A` continues to exist on worker 2 even after the call `remotecall_fetch` returns. Remote calls with embedded global references (under `Main` module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.
- Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes. For example:

  ```
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 2) # worker 2
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 3) # worker 3
  A = nothing
  ```

  Executing the above snippet results in `Main.A` on worker 2 having a different value from `Main.A` on worker 3, while the value of `Main.A` on node 1 is set to `nothing`.

As you may have realized, while memory associated with globals may be collected when they are reassigned on the master, no such action is taken on the workers as the bindings continue to be valid. `clear!` can be used to manually reassign specific globals on remote nodes to `nothing` once they are no longer required. This will release any memory associated with them as part of a regular garbage collection cycle.

Thus programs should be careful referencing globals in remote calls. In fact, it is preferable to avoid them altogether if possible. If you must reference globals, consider using `let` blocks to localize global variables.

For example:

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);

julia> B = rand(10,10);

julia> let B = B
           remotecall_fetch(()->B, 2)
       end;

julia> @fetchfrom 2 InteractiveUtils.varinfo()
name           size summary
———————— ———————— ———————————————————
A         800 bytes 10×10 Array{Float64,2}
Base                Module
Core                Module
Main                Module
```

As can be seen, global variable `A` is defined on worker 2, but `B` is captured as a local variable and hence a binding for `B` does not exist on worker 2.

28.5 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a Monte Carlo simulation, where multiple processes can handle independent simulation trials simultaneously. We can use `@spawnat` to flip coins on two processes. First, write the following function in `count_heads.jl`:

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

The function `count_heads` simply adds together `n` random bits. Here is how we can perform some trials on two machines, and add together the results:

```
julia> @everywhere include_string(Main, $(read("count_heads.jl", String)), "count_heads.jl")
```

```
julia> a = @spawnat :any count_heads(100000000)
Future(2, 1, 6, nothing)

julia> b = @spawnat :any count_heads(100000000)
Future(3, 1, 7, nothing)

julia> fetch(a)+fetch(b)
100001564
```

This example demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processes, and then their results are combined using some function. The combination process is called a reduction, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern `x = f(x,v[i])`, where `x` is the accumulator, `f` is the reduction function, and the `v[i]` are the elements being reduced. It is desirable for `f` to be associative, so that it does not matter what order the operations are performed in.

Notice that our use of this pattern with `count_heads` can be generalized. We used two explicit `@spawnat` statements, which limits the parallelism to two processes. To run on any number of processes, we can use a parallel for loop, running in distributed memory, which can be written in Julia using `@distributed` like this:

```
nheads = @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end
```

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case `(+)`). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```
a = zeros(100000)
@distributed for i = 1:100000
    a[i] = i
end
```

This code will not initialize all of `a`, since each process will have a separate copy of it. Parallel for loops like these must be avoided. Fortunately, Shared Arrays can be used to get around this limitation:

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)
@distributed (+) for i = 1:100000
    f(a[rand(1:end)])
end
```

Here each iteration applies `f` to a randomly-chosen sample from a vector `a` shared by all processes.

As you could see, the reduction operator can be omitted if it is not needed. In that case, the loop executes asynchronously, i.e. it spawns independent tasks on all available workers and returns an array of `Future` immediately without waiting for completion. The caller can wait for the `Future` completions at a later point by calling `fetch` on them, or wait for completion at the end of the loop by prefixing it with `@sync`, like `@sync @distributed for`.

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called parallel map, implemented in Julia as the `pmap` function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];

julia> pmap(svdvals, M);
```

Julia's `pmap` is designed for the case where each function call does a large amount of work. In contrast, `@distributed for` can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both `pmap` and `@distributed for` for the parallel computation. In case of `@distributed for`, the final reduction is done on the calling process.

28.6 Remote References and AbstractChannels

Remote references always refer to an implementation of an `AbstractChannel`.

A concrete implementation of an `AbstractChannel` (like `Channel`), is required to implement `put!`, `take!`, `fetch`, `isready` and `wait`. The remote object referred to by a `Future` is stored in a `Channel{Any}(1)`, i.e., a `Channel` of size 1 capable of holding objects of `Any` type.

`RemoteChannel`, which is rewritable, can point to any type and size of channels, or any other implementation of an `AbstractChannel`.

The constructor `RemoteChannel(f::Function, pid)()` allows us to construct references to channels holding more than one value of a specific type. `f` is a function executed on `pid` and it must return an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10. The channel exists on worker `pid`.

Methods `put!`, `take!`, `fetch`, `isready` and `wait` on a `RemoteChannel` are proxied onto the backing store on the remote process.

`RemoteChannel` can thus be used to refer to user implemented `AbstractChannel` objects. A simple example of this is provided in `dictchannel.jl` in the Examples repository, which uses a dictionary as its remote store.

28.7 Channels and RemoteChannels

- A `Channel` is local to a process. Worker 2 cannot directly refer to a `Channel` on worker 3 and vice-versa. A `RemoteChannel`, however, can put and take values across workers.
- A `RemoteChannel` can be thought of as a handle to a `Channel`.
- The process id, `pid`, associated with a `RemoteChannel` identifies the process where the backing store, i.e., the backing `Channel` exists.
- Any process with a reference to a `RemoteChannel` can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a `RemoteChannel` is associated with.
- Serializing a `Channel` also serializes any data present in the channel. Deserializing it therefore effectively makes a copy of the original object.
- On the other hand, serializing a `RemoteChannel` only involves the serialization of an identifier that identifies the location and instance of `Channel` referred to by the handle. A deserialized `RemoteChannel` object (on any worker), therefore also points to the same backing store as the original.

The channels example from above can be modified for interprocess communication, as shown below.

We start 4 workers to process a single **jobs** remote channel. Jobs, identified by an id (**job_id**), are written to the channel. Each remotely executing task in this simulation reads a **job_id**, waits for a random amount of time and writes back a tuple of **job_id**, time taken and its own **pid** to the results channel. Finally all the **results** are printed out on the master process.

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));

julia> @everywhere function do_work(jobs, results) # define work function everywhere
           while true
               job_id = take!(jobs)
               exec_time = rand()
               sleep(exec_time) # simulates elapsed time doing actual work
               put!(results, (job_id, exec_time, myid()))
           end
       end

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> @async make_jobs(n); # feed the jobs channel with "n" jobs

julia> for p in workers() # start tasks on the workers to process requests in parallel
           remote_do(do_work, p, jobs, results)
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time, where = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds on worker $where")
           global n = n - 1
```

```
    end
1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
10 finished in 0.58 seconds on worker 2
0.055971741
```

Remote References and Distributed Garbage Collection

Objects referred to by remote references can be freed only when all held references in the cluster are deleted.

The node where the value is stored keeps track of which of the workers have a reference to it. Every time a `RemoteChannel` or a (unfetched) `Future` is serialized to a worker, the node pointed to by the reference is notified. And every time a `RemoteChannel` or a (unfetched) `Future` is garbage collected locally, the node owning the value is again notified. This is implemented in an internal cluster aware serializer. Remote references are only valid in the context of a running cluster. Serializing and deserializing references to and from regular `IO` objects is not supported.

The notifications are done via sending of "tracking" messages—an "add reference" message when a reference is serialized to a different process and a "delete reference" message when a reference is locally garbage collected.

Since `Future`s are write-once and cached locally, the act of `fetch`ing a `Future` also updates reference tracking information on the node owning the value.

The node which owns the value frees it once all references to it are cleared.

With `Future`s, serializing an already fetched `Future` to a different node also sends the value since the original remote store may have collected the value by this time.

It is important to note that when an object is locally garbage collected depends on the size of the object and the current memory pressure in the system.

In case of remote references, the size of the local reference object is quite small, while the value stored on the remote node may be quite large. Since the local object may not be collected immediately, it is a good practice to explicitly call `finalize` on local instances of a `RemoteChannel`, or on unfetched `Future`s. Since calling `fetch` on a `Future` also removes

its reference from the remote store, this is not required on fetched Futures. Explicitly calling finalize results in an immediate message sent to the remote node to go ahead and remove its reference to the value.

Once finalized, a reference becomes invalid and cannot be used in any further calls.

28.8 Local invocations

Data is necessarily copied over to the remote node for execution. This is the case for both remotecalls and when data is stored to aRemoteChannel / Future on a different node. As expected, this results in a copy of the serialized objects on the remote node. However, when the destination node is the local node, i.e. the calling process id is the same as the remote node id, it is executed as a local call. It is usually (not always) executed in a different task - but there is no serialization/deserialization of data. Consequently, the call refers to the same object instances as passed - no copies are created. This behavior is highlighted below:

```
julia> using Distributed;

julia> rc = RemoteChannel(()->Channel(3));   # RemoteChannel created on local node

julia> v = [0];

julia> for i in 1:3
           v[1] = i                          # Reusing `v`
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[3], [3], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 1

julia> addprocs(1);

julia> rc = RemoteChannel(()->Channel(3), workers()[1]);   # RemoteChannel created on remote node

julia> v = [0];

julia> for i in 1:3
```

```
            v[1] = i
            put!(rc, v)
        end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[1], [2], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 3
```

As can be seen, `put!` on a locally owned `RemoteChannel` with the same object `v` modifed between calls results in the same single object instance stored. As opposed to copies of `v` being created when the node owning `rc` is a different node.

It is to be noted that this is generally not an issue. It is something to be factored in only if the object is both being stored locally and modifed post the call. In such cases it may be appropriate to store a `deepcopy` of the object.

This is also true for remotecalls on the local node as seen in the following example:

```
julia> using Distributed; addprocs(1);

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), myid(), v);     # Executed on local node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[1], v2=[1], true

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), workers()[1], v); # Executed on remote node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[0], v2=[1], false
```

As can be seen once again, a remote call onto the local node behaves just like a direct invocation. The call modifies local objects passed as arguments. In the remote invocation, it operates on a copy of the arguments.

To repeat, in general this is not an issue. If the local node is also being used as a compute node, and the arguments used post the call, this behavior needs to be factored in and if required deep copies of arguments must be passed to the call invoked on the local node. Calls on remote nodes will always operate on copies of arguments.

28.9 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a `DArray`, the behavior of a `SharedArray` is quite different. In a `DArray`, each process has local access to just a chunk of the data, and no two processes share the same chunk; in contrast, in a `SharedArray` each "participating" process has access to the entire array. A `SharedArray` is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

Shared Array support is available via module `SharedArrays` which must be explicitly loaded on all participating workers.

`SharedArray` indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on `SharedArray`s, albeit in single-process mode. In cases where an algorithm insists on an `Array` input, the underlying array can be retrieved from a `SharedArray` by calling `sdata`. For other `AbstractArray` types, `sdata` just returns the object itself, so it's safe to use `sdata` on any `Array`-type object.

The constructor for a shared array is of the form:

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

which creates an `N`-dimensional shared array of a bits type `T` and size `dims` across the processes specified by `pids`. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `pids` named argument (and the creating process too, if it is on the same host). Note that only elements that are `isbits` are supported in a SharedArray.

If an `init` function, of signature `initfn(S::SharedArray)`, is specified, it is called on all the participating workers. You can specify that each worker runs the `init` function on a distinct portion of the array, thereby parallelizing initialization.

Here's a brief example:

```
julia> using Distributed


julia> addprocs(3)
3-element Array{Int64,1}:
```

```
 2
 3
 4

julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S -> S[localindices(S)] = repeat([myid()], length(localindices(S))))
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

SharedArrays.localindices provides disjoint one-dimensional ranges of indices, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

```
julia> S = SharedArray{Int,2}((3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] = repeat([myid()],
↪  length( indexpids(S):length(procs(S)):length(S))))
3×4 SharedArray{Int64,2}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
@sync begin
    for p in procs(S)
        @async begin
            remotecall_wait(fill!, p, S, p)
        end
    end
end
```

would result in undefined behavior. Because each process fills the entire array with its own `pid`, whichever process is the last to execute (for any particular element of `S`) will have its `pid` retained.

As a more extended and complex example, consider running the following "kernel" in parallel:

```
q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if `q[i,j,t]` is near the end of the block assigned to one worker and `q[i,j,t+1]` is near the beginning of the block assigned to another, it's very likely that `q[i,j,t]` will not be ready at the time it's needed for computing `q[i,j,t+1]`. In such cases, one is better off chunking the array manually. Let's split along the second dimension. Define a function that returns the `(irange, jrange)` indices assigned to this worker:

```
julia> @everywhere function myrange(q::SharedArray)
           idx = indexpids(q)
           if idx == 0 # This worker is not assigned a piece
               return 1:0, 1:0
           end
           nchunks = length(procs(q))
           splits = [round(Int, s) for s in range(0, stop=size(q,2), length=nchunks+1)]
           1:size(q,1), splits[idx]+1:splits[idx+1]
       end
```

Next, define the kernel:

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange, trange)
           @show (irange, jrange, trange)  # display so we can see what's happening
           for t in trange, j in jrange, i in irange
               q[i,j,t+1] = q[i,j,t] + u[i,j,t]
           end
           q
       end
```

We also define a convenience wrapper for a `SharedArray` implementation

```
julia> @everywhere advection_shared_chunk!(q, u) =
           advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

Now let's compare three different versions, one that runs in a single process:

```
julia> advection_serial!(q, u) = advection_chunk!(q, u, 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

one that uses `@distributed`:

```
julia> function advection_parallel!(q, u)
           for t = 1:size(q,3)-1
               @sync @distributed for j = 1:size(q,2)
                   for i = 1:size(q,1)
                       q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                   end
               end
           end
           q
       end;
```

and one that delegates in chunks:

```
julia> function advection_shared!(q, u)
           @sync begin
               for p in procs(q)
                   @async remotecall_wait(advection_shared_chunk!, p, q, u)
               end
           end
           q
       end;
```

If we create `SharedArray`s and time these functions, we get the following results (with `julia -p 4`):

```
julia> q = SharedArray{Float64,3}((500,500,500));

julia> u = SharedArray{Float64,3}((500,500,500));
```

Run the functions once to JIT-compile and `@time` them on the second run:

```
julia> @time advection_serial!(q, u);
(irange,jrange,trange) = (1:500,1:500,1:499)
 830.220 milliseconds (216 allocations: 13820 bytes)
```

```
julia> @time advection_parallel!(q, u);
   2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)

julia> @time advection_shared!(q,u);
        From worker 2:       (irange,jrange,trange) = (1:500,1:125,1:499)
        From worker 4:       (irange,jrange,trange) = (1:500,251:375,1:499)
        From worker 3:       (irange,jrange,trange) = (1:500,126:250,1:499)
        From worker 5:       (irange,jrange,trange) = (1:500,376:500,1:499)
 238.119 milliseconds (2264 allocations: 169 KB)
```

The biggest advantage of `advection_shared!` is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

Shared Arrays and Distributed Garbage Collection

Like remote references, shared arrays are also dependent on garbage collection on the creating node to release references from all participating workers. Code which creates many short lived shared array objects would benefit from explicitly finalizing these objects as soon as possible. This results in both memory and file handles mapping the shared segment being released sooner.

28.10 ClusterManagers

The launching, management and networking of Julia processes into a logical cluster is done via cluster managers. A `ClusterManager` is responsible for

- launching worker processes in a cluster environment
- managing events during the lifetime of each worker
- optionally, providing data transport

A Julia cluster has the following characteristics:

- The initial Julia process, also called the `master`, is special and has an `id` of 1.
- Only the `master` process can add or remove worker processes.
- All processes can directly communicate with each other.

Connections between workers (using the in-built TCP/IP transport) is established in the following manner:

- `addprocs` is called on the master process with a `ClusterManager` object.
- `addprocs` calls the appropriate `launch` method which spawns required number of worker processes on appropriate machines.
- Each worker starts listening on a free port and writes out its host and port information to `stdout`.
- The cluster manager captures the `stdout` of each worker and makes it available to the master process.
- The master process parses this information and sets up TCP/IP connections to each worker.
- Every worker is also notified of other workers in the cluster.
- Each worker connects to all workers whose `id` is less than the worker's own `id`.
- In this way a mesh network is established, wherein every worker is directly connected with every other worker.

While the default transport layer uses plain `TCPSocket`, it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

- `LocalManager`, used when `addprocs()` or `addprocs(np::Integer)` are called
- `SSHManager`, used when `addprocs(hostnames::Array)` is called with a list of hostnames

`LocalManager` is used to launch additional workers on the same host, thereby leveraging multi-core and multi-processor hardware.

Thus, a minimal cluster manager would need to:

- be a subtype of the abstract `ClusterManager`
- implement `launch`, a method responsible for launching new workers
- implement `manage`, which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

`addprocs(manager::FooManager)` requires `FooManager` to implement:

```
function launch(manager::FooManager, params::Dict, launched::Array, c::Condition)
    [...]
end
```

```
function manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

As an example let us see how the `LocalManager`, the manager responsible for starting workers on the same host, is implemented:

```
struct LocalManager <: ClusterManager
    np::Integer
end

function launch(manager::LocalManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::LocalManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

The launch method takes the following arguments:

- `manager::ClusterManager`: the cluster manager that addprocs is called with
- `params::Dict`: all the keyword arguments passed to addprocs
- `launched::Array`: the array to append one or more `WorkerConfig` objects to
- `c::Condition`: the condition variable to be notified as and when workers are launched

The launch method is called asynchronously in a separate task. The termination of this task signals that all requested workers have been launched. Hence the launch function MUST exit as soon as all the requested workers have been launched.

Newly launched workers are connected to each other and the master process in an all-to-all manner. Specifying the command line argument `--worker[=<cookie>]` results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets.

All workers in a cluster share the same cookie as the master. When the cookie is unspecified, i.e, with the `--worker` option, the worker tries to read it from its standard input. `LocalManager` and `SSHManager` both pass the cookie to newly launched workers via their standard inputs.

By default a worker will listen on a free port at the address returned by a call to `getipaddr()`. A specific address to listen on may be specified by optional argument `--bind-to bind_addr[:port]`. This is useful for multi-homed hosts.

As an example of a non-TCP/IP transport, an implementation may choose to use MPI, in which case `--worker` must NOT be specified. Instead, newly launched workers should call `init_worker(cookie)` before using any of the parallel constructs.

For every worker launched, the `launch` method must add a `WorkerConfig` object (with appropriate fields initialized) to `launched`

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
    io::Union{IO, Nothing}
    host::Union{AbstractString, Nothing}
    port::Union{Integer, Nothing}

    # Used when launching additional workers at a host
    count::Union{Int, Symbol, Nothing}
    exename::Union{AbstractString, Cmd, Nothing}
    exeflags::Union{Cmd, Nothing}

    # External cluster managers can use this to store information at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Any

    # SSHManager / SSH tunnel connections to workers
    tunnel::Union{Bool, Nothing}
    bind_addr::Union{AbstractString, Nothing}
    sshflags::Union{Cmd, Nothing}
    max_parallel::Union{Integer, Nothing}

    # Used by Local/SSH managers
    connect_at::Any

    [...]
end
```

Most of the fields in `WorkerConfig` are used by the inbuilt managers. Custom cluster managers would typically specify only `io` or `host` / `port`:

- If `io` is specified, it is used to read host/port information. A Julia worker prints out its bind address and port at startup. This allows Julia workers to listen on any free port available instead of requiring worker ports to be configured manually.
- If `io` is not specified, `host` and `port` are used to connect.
- `count`, `exename` and `exeflags` are relevant for launching additional workers from a worker. For example, a cluster manager may launch a single worker per node, and use that to launch additional workers.
  - `count` with an integer value `n` will launch a total of `n` workers.
  - `count` with a value of `:auto` will launch as many workers as the number of CPU threads (logical cores) on that machine.
  - `exename` is the name of the `julia` executable including the full path.
  - `exeflags` should be set to the required command line arguments for new workers.
- `tunnel`, `bind_addr`, `sshflags` and `max_parallel` are used when a ssh tunnel is required to connect to the workers from the master process.
- `userdata` is provided for custom cluster managers to store their own worker-specific information.

`manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)` is called at different times during the worker's lifetime with appropriate `op` values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

Cluster Managers with Custom Transports

Replacing the default TCP/IP all-to-all socket connections with a custom transport layer is a little more involved. Each Julia process has as many communication tasks as the workers it is connected to. For example, consider a Julia cluster of 32 processes in an all-to-all mesh network:

- Each Julia process thus has 31 communication tasks.
- Each task handles all incoming messages from a single remote worker in a message-processing loop.

- The message-processing loop waits on an `IO` object (for example, a `TCPSocket` in the default implementation), reads an entire message, processes it and waits for the next one.
- Sending messages to a process is done directly from any Julia task—not just communication tasks—again, via the appropriate `IO` object.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate `IO` objects that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

The default implementation (which uses TCP/IP sockets) is implemented as `connect(manager::ClusterManager, pid::Integer, config::WorkerConfig)`.

`connect` should return a pair of `IO` objects, one for reading data sent from worker `pid`, and the other to write data that needs to be sent to worker `pid`. Custom cluster managers can use an in-memory `BufferStream` as the plumbing to proxy data between the custom, possibly non-`IO` transport and Julia's in-built parallel infrastructure.

A `BufferStream` is an in-memory `IOBuffer` which behaves like an `IO`—it is a stream which can be handled asynchronously.

The folder `clustermanager/0mq` in the Examples repository contains an example of using ZeroMQ to connect Julia workers in a star topology with a 0MQ broker in the middle. Note: The Julia processes are still all logically connected to each other—any worker can message any other worker directly without any awareness of 0MQ being used as the transport layer.

When using custom transports:

- Julia workers must NOT be started with `--worker`. Starting with `--worker` will result in the newly launched workers defaulting to the TCP/IP socket transport implementation.
- For every incoming logical connection with a worker, `Base.process_messages(rd::IO, wr::IO)()` must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the `IO` objects.
- `init_worker(cookie, manager::FooManager)` must be called as part of worker process initialization.
- Field `connect_at::Any` in `WorkerConfig` can be set by the cluster manager when `launch` is called. The value of this field is passed in all `connect` callbacks. Typically, it carries information on how to connect to a worker.

For example, the TCP/IP socket transport uses this field to specify the `(host, port)` tuple at which to connect to a worker.

`kill(manager, pid, config)` is called to remove a worker from the cluster. On the master process, the corresponding `IO` objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an `exit()` call on the specified remote worker.

The Examples folder `clustermanager/simple` is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

Network Requirements for LocalManager and SSHManager

Julia clusters are designed to be executed on already secured environments on infrastructure such as local laptops, departmental clusters, or even the cloud. This section covers network security requirements for the inbuilt `LocalManager` and `SSHManager`:

- The master process does not listen on any port. It only connects out to the workers.
- Each worker binds to only one of the local interfaces and listens on an ephemeral port number assigned by the OS.
- `LocalManager`, used by `addprocs(N)`, by default binds only to the loopback interface. This means that workers started later on remote hosts (or by anyone with malicious intentions) are unable to connect to the cluster. An `addprocs(4)` followed by an `addprocs(["remote_host"])` will fail. Some users may need to create a cluster comprising their local system and a few remote systems. This can be done by explicitly requesting `LocalManager` to bind to an external network interface via the `restrict` keyword argument: `addprocs(4; restrict=false)`.
- `SSHManager`, used by `addprocs(list_of_remote_hosts)`, launches workers on remote hosts via SSH. By default SSH is only used to launch Julia workers. Subsequent master-worker and worker-worker connections use plain, unencrypted TCP/IP sockets. The remote hosts must have passwordless login enabled. Additional SSH flags or credentials may be specified via keyword argument `sshflags`.
- `addprocs(list_of_remote_hosts; tunnel=true, sshflags=<ssh keys and other flags>)` is useful when we wish to use SSH connections for master-worker too. A typical scenario for this is a local laptop running the Julia REPL (i.e., the master) with the rest of the cluster on the cloud, say on Amazon EC2. In this case only port 22 needs to be opened at the remote cluster coupled with SSH client authenticated via public key infrastructure (PKI). Authentication credentials can be supplied via `sshflags`, for example `sshflags=`-i <keyfile>`.

  In an all-to-all topology (the default), all workers connect to each other via plain TCP sockets. The security policy on the cluster nodes must thus ensure free connectivity between workers for the ephemeral port range (varies by OS).

Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom `ClusterManager`.

- If you specify `multiplex=true` as an option to `addprocs`, SSH multiplexing is used to create a tunnel between the master and workers. If you have configured SSH multiplexing on your own and the connection has already been established, SSH multiplexing is used regardless of `multiplex` option. If multiplexing is enabled, forwarding is set by using the existing connection (`-O forward` option in ssh). This is beneficial if your servers require password authentication; you can avoid authentication in Julia by logging in to the server ahead of `addprocs`. The control socket will be located at `~/.ssh/julia-%r@%h:%p` during the session unless the existing multiplexing connection is used. Note that bandwidth may be limited if you create multiple processes on a node and enable multiplexing, because in that case processes share a single multiplexing TCP connection.

Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

- `cluster_cookie()` returns the cookie, while `cluster_cookie(cookie)()` sets it and returns the new cookie.
- All connections are authenticated on both sides to ensure that only workers started by the master are allowed to connect to each other.
- The cookie may be passed to the workers at startup via argument `--worker=<cookie>`. If argument `--worker` is specified without the cookie, the worker tries to read the cookie from its standard input (`stdin`). The `stdin` is closed immediately after the cookie is retrieved.
- `ClusterManager`s can retrieve the cookie on the master by calling `cluster_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `--worker`) must call `init_worker(cookie, manager)` with the same cookie as on the master.

Note that environments requiring higher levels of security can implement this via a custom `ClusterManager`. For example, cookies can be pre-shared and hence not specified as a startup argument.

28.11 Specifying Network Topology (Experimental)

The keyword argument `topology` passed to `addprocs` is used to specify how the workers must be connected to each other:

- `:all_to_all`, the default: all workers are connected to each other.

- `:master_worker`: only the driver process, i.e. `pid` 1, has connections to the workers.
- `:custom`: the `launch` method of the cluster manager specifies the connection topology via the fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster-manager-provided identity `ident` will connect to all workers specified in `connect_idents`.

Keyword argument `lazy=true¦false` only affects `topology` option `:all_to_all`. If `true`, the cluster starts off with the master connected to all workers. Specific worker-worker connections are established at the first remote invocation between two workers. This helps in reducing initial resources allocated for intra-cluster communication. Connections are setup depending on the runtime requirements of a parallel program. Default value for `lazy` is `true`.

Currently, sending a message between unconnected workers results in an error. This behaviour, as with the functionality and interface, should be considered experimental in nature and may change in future releases.

28.12 Noteworthy external packages

Outside of Julia parallelism there are plenty of external packages that should be mentioned. For example MPI.jl is a Julia wrapper for the `MPI` protocol, or DistributedArrays.jl, as presented in Shared Arrays. A mention must be made of Julia's GPU programming ecosystem, which includes:

1. Low-level (C kernel) based operations OpenCL.jl and CUDAdrv.jl which are respectively an OpenCL interface and a CUDA wrapper.
2. Low-level (Julia Kernel) interfaces like CUDAnative.jl which is a Julia native CUDA implementation.
3. High-level vendor-specific abstractions like CuArrays.jl and CLArrays.jl
4. High-level libraries like ArrayFire.jl and GPUArrays.jl

In the following example we will use both `DistributedArrays.jl` and `CuArrays.jl` to distribute an array across multiple processes by first casting it through `distribute()` and `CuArray()`.

Remember when importing `DistributedArrays.jl` to import it across all processes using `@everywhere`

```
$ ./julia -p 4

julia> addprocs()

julia> @everywhere using DistributedArrays
```

```
julia> using CuArrays

julia> B = ones(10_000) ./ 2;

julia> A = ones(10_000) .* π;

julia> C = 2 .* A ./ B;

julia> all(C .≈ 4*π)
true

julia> typeof(C)
Array{Float64,1}

julia> dB = distribute(B);

julia> dA = distribute(A);

julia> dC = 2 .* dA ./ dB;

julia> all(dC .≈ 4*π)
true

julia> typeof(dC)
DistributedArrays.DArray{Float64,1,Array{Float64,1}}

julia> cuB = CuArray(B);

julia> cuA = CuArray(A);

julia> cuC = 2 .* cuA ./ cuB;

julia> all(cuC .≈ 4*π);
true

julia> typeof(cuC)
CuArray{Float64,1}
```

Keep in mind that some Julia features are not currently supported by CUDAnative.jl[2] , especially some functions like `sin` will need to be replaced with `CUDAnative.sin`(cc: @maleadt).

In the following example we will use both `DistributedArrays.jl` and `CuArrays.jl` to distribute an array across multiple processes and call a generic function on it.

```
function power_method(M, v)
    for i in 1:100
        v = M*v
        v /= norm(v)
    end

    return v, norm(M*v) / norm(v)  # or  (M*v) ./ v
end
```

`power_method` repeatedly creates a new vector and normalizes it. We have not specified any type signature in function declaration, let's see if it works with the aforementioned datatypes:

```
julia> M = [2. 1; 1 1];

julia> v = rand(2)
2-element Array{Float64,1}:
0.40395
0.445877

julia> power_method(M,v)
([0.850651, 0.525731], 2.618033988749895)

julia> cuM = CuArray(M);

julia> cuv = CuArray(v);

julia> curesult = power_method(cuM, cuv);

julia> typeof(curesult)
CuArray{Float64,1}

julia> dM = distribute(M);

julia> dv = distribute(v);

julia> dC = power_method(dM, dv);
```

```
julia> typeof(dC)
Tuple{DistributedArrays.DArray{Float64,1,Array{Float64,1}},Float64}
```

To end this short exposure to external packages, we can consider `MPI.jl`, a Julia wrapper of the MPI protocol. As it would take too long to consider every inner function, it would be better to simply appreciate the approach used to implement the protocol.

Consider this toy script which simply calls each subprocess, instantiate its rank and when the master process is reached, performs the ranks' sum

```
import MPI

MPI.Init()

comm = MPI.COMM_WORLD
MPI.Barrier(comm)

root = 0
r = MPI.Comm_rank(comm)

sr = MPI.Reduce(r, MPI.SUM, root, comm)

if(MPI.Comm_rank(comm) == root)
   @printf("sum of ranks: %s\n", sr)
end

MPI.Finalize()
```

```
mpirun -np 4 ./julia example.jl
```

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding rma to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see https://mpi-forum.org/docs.

[2] Julia GPU man pages

Chapter 29

Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```
julia> `echo hello`
`echo hello`
```

differs in several aspects from the behavior in various shells, Perl, or Ruby:

- Instead of immediately running the command, backticks create a `Cmd` object to represent the command. You can use this object to connect the command to others via pipes, `run` it, and `read` or `write` to it.
- When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to `stdout` as it would using `libc`'s `system` call.
- The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as `julia`'s immediate child process, using `fork` and `exec` calls.

Here's a simple example of running an external program:

```
julia> mycommand = `echo hello`
`echo hello`

julia> typeof(mycommand)
Cmd

julia> run(mycommand);
hello
```

The `hello` is the output of the `echo` command, sent to `stdout`. The run method itself returns `nothing`, and throws an `ErrorException` if the external command fails to run successfully.

If you want to read the output of the external command, `read` can be used instead:

```
julia> a = read(`echo hello`, String)
"hello\n"

julia> chomp(a) == "hello"
true
```

More generally, you can use `open` to read from or write to an external command.

```
julia> open(`less`, "w", stdout) do io
           for i = 1:3
               println(io, i)
           end
       end
1
2
3
```

The program name and the individual arguments in a command can be accessed and iterated over as if the command were an array of strings:

```
julia> collect(`echo "foo bar"`)
2-element Array{String,1}:
 "echo"
 "foo bar"

julia> `echo "foo bar"`[2]
"foo bar"
```

29.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use $ for interpolation much as you would in a string literal (see Strings):

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted to sort the contents of the file `/Volumes/External HD/data.csv`. Let's try it:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

How did the file name get quoted? Julia knows that `file` is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of `file` is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"

julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

As you can see, the space in the `path` variable is appropriately escaped. But what if you want to interpolate multiple words? In that case, just use an array (or any other iterable container):

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element Array{String,1}:
```

```
 "/etc/passwd"
 "/Volumes/External HD/data.csv"

julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

If you interpolate an array as part of a shell word, Julia emulates the shell's `{a,b,c}` argument generation:

```
julia> names = ["foo","bar","baz"]
3-element Array{String,1}:
 "foo"
 "bar"
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

```
julia> names = ["foo","bar","baz"]
3-element Array{String,1}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Array{String,1}:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
julia> `rm -rf $["foo","bar","baz","qux"].$["aux","log","pdf"]`
`rm -rf foo.aux foo.log foo.pdf bar.aux bar.log bar.pdf baz.aux baz.log baz.pdf qux.aux qux.log qux.pdf`
```

29.2 Quoting

Inevitably, one wants to write commands that aren't quite so simple, and it becomes necessary to use quotes. Here's a simple example of a Perl one-liner at a shell prompt:

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

The Perl expression needs to be in single quotes for two reasons: so that spaces don't break the expression into multiple shell words, and so that uses of Perl variables like $| (yes, that's the name of a variable in Perl), don't cause interpolation. In other instances, you may want to use double quotes so that interpolation does occur:

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

In general, the Julia backtick syntax is carefully designed so that you can just cut-and-paste shell commands as is into backticks and they will work: the escaping, quoting, and interpolation behaviors are the same as the shell's. The only difference is that the interpolation is integrated and aware of Julia's notion of what is a single string value, and what is a container for multiple values. Let's try the above two examples in Julia:

```
julia> A = `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(A);
0
1
2
3

julia> first = "A"; second = "B";

julia> B = `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(B);
```

```
1: A
2: B
```

The results are identical, and Julia's interpolation behavior mimics the shell's with some improvements due to the fact that Julia supports first-class iterable objects while most shells use strings split on spaces for this, which introduces ambiguities. When trying to port shell commands to Julia, try cut and pasting first. Since Julia shows commands to you before running them, you can easily and safely just examine its interpretation without doing any damage.

29.3 Pipelines

Shell metacharacters, such as `|`, `&`, and `>`, need to be quoted (or escaped) inside of Julia's backticks:

```
julia> run(`echo hello '|' sort`);
hello | sort

julia> run(`echo hello \| sort`);
hello | sort
```

This expression invokes the `echo` command with three words as arguments: `hello`, `|`, and `sort`. The result is that a single line is printed: `hello | sort`. How, then, does one construct a pipeline? Instead of using `'|'` inside of backticks, one uses `pipeline`:

```
julia> run(pipeline(`echo hello`, `sort`));
hello
```

This pipes the output of the `echo` command to the `sort` command. Of course, this isn't terribly interesting since there's only one line to sort, but we can certainly do much more interesting things:

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail -n5`))
210
211
212
213
214
```

This prints the highest five user IDs on a UNIX system. The `cut`, `sort` and `tail` commands are all spawned as immediate children of the current `julia` process, with no intervening shell process. Julia itself does the work to setup pipes and

connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

```
julia> run(`echo hello` & `echo world`);
world
hello
```

The order of the output here is non-deterministic because the two `echo` processes are started nearly simultaneously, and race to make the first write to the `stdout` descriptor they share with each other and the `julia` parent process. Julia lets you pipe the output from both of these processes to another program:

```
julia> run(pipeline(`echo world` & `echo hello`, `sort`));
hello
world
```

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both `echo` processes, and the other end of the pipe is read from by the `sort` command.

IO redirection can be accomplished by passing keyword arguments `stdin`, `stdout`, and `stderr` to the `pipeline` function:

```
pipeline(`do_work`, stdout=pipeline(`sort`, "out.txt"), stderr="errs.txt")
```

Avoiding Deadlock in Pipelines

When reading and writing to both ends of a pipeline from a single process, it is important to avoid forcing the kernel to buffer all of the data.

For example, when reading all of the output from a command, call `read(out, String)`, not `wait(process)`, since the former will actively consume all of the data written by the process, whereas the latter will attempt to store the data in the kernel's buffers while waiting for a reader to be connected.

Another common solution is to separate the reader and writer of the pipeline into separate `Task`s:

```
writer = @async write(process, "data")
reader = @async do_compute(read(process, String))
wait(process)
fetch(reader)
```

Complex Example

The combination of a high-level programming language, a first-class command abstraction, and automatic setup of pipes between processes is a powerful one. To give some sense of the complex pipelines that can be created easily, here are some more sophisticated examples, with apologies for the excessive use of Perl one-liners:

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '$sleep';'`;

julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`, prefixer("A",2) & prefixer("B",2)));
B 0
A 1
B 2
A 3
B 4
A 5
```

This is a classic example of a single producer feeding two concurrent consumers: one `perl` process generates lines with the numbers 0 through 5 on them, while two parallel processes consume that output, one prefixing lines with the letter "A", the other with the letter "B". Which consumer gets the first line is non-deterministic, but once that race has been won, the lines are consumed alternately by one process and then the other. (Setting `$|=1` in Perl causes each print statement to flush the `stdout` handle, which is necessary for this example to work. Otherwise all the output is buffered and printed to the pipe at once, to be read by just one consumer process.)

Here is an even more complex multi-stage producer-consumer example:

```
julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`,
           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3),
           prefixer("A",2) & prefixer("B",2)));
A X 0
B Y 1
A Z 2
B X 3
A Y 4
B Z 5
```

This example is similar to the previous one, except there are two stages of consumers, and the stages have different latency so they use a different number of parallel workers, to maintain saturated throughput.

We strongly encourage you to try all these examples to see how they work.

Chapter 30

Calling C and Fortran Code

Though most code can be written in Julia, there are many high-quality, mature libraries for numerical computing already written in C and Fortran. To allow easy use of this existing code, Julia makes it simple and efficient to call C and Fortran functions. Julia has a "no boilerplate" philosophy: functions can be called directly from Julia without any "glue" code, code generation, or compilation – even from the interactive prompt. This is accomplished just by making an appropriate call with `ccall` syntax, which looks like an ordinary function call.

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the `-shared` and `-fPIC` options. The machine instructions generated by Julia's JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. [1]

Shared libraries and functions are referenced by a tuple of the form `(:function, "library")` or `("function", "library")` where `function` is the C-exported function name, and `library` refers to the shared library name. Shared libraries available in the (platform-specific) load path will be resolved by name. The full path to the library may also be specified.

A function name may be used alone in place of the tuple (just `:function` or `"function"`). In this case the name is resolved within the current process. This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia.

By default, Fortran compilers generate mangled names (for example, converting function names to lowercase or uppercase, often appending an underscore), and so to call a Fortran function via `ccall` you must pass the mangled identifier corresponding to the rule followed by your Fortran compiler. Also, when calling a Fortran function, all inputs must be passed as pointers to allocated values on the heap or stack. This applies not only to arrays and other mutable objects which are normally heap-allocated, but also to scalar values such as integers and floats which are normally stack-allocated and commonly passed in registers when using C or Julia calling conventions.

Finally, you can use `ccall` to actually generate a call to the library function. The arguments to `ccall` are:

1. A `(:function, "library")` pair (most common),

   OR

   a `:function` name symbol or `"function"` name string (for symbols in the current process or libc),

   OR

   a function pointer (for example, from `dlsym`).

2. The function's return type

3. A tuple of input types, corresponding to the function signature

4. The actual argument values to be passed to the function, if any; each is a separate parameter.

   Note

   The `(:function, "library")` pair, return type, and input types must be literal constants (i.e., they can't be variables, but see Non-constant Function Specifications below).

   The remaining parameters are evaluated at compile time, when the containing method is defined.

   Note

   See below for how to map C types to Julia types.

As a complete but simple example, the following calls the `clock` function from the standard C library on most Unix-derived systems:

```
julia> t = ccall(:clock, Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` takes no arguments and returns an `Int32`. One common gotcha is that a 1-tuple of argument types must be written with a trailing comma. For example, to call the `getenv` function to get a pointer to the value of an environment variable, one makes a call like this:

```
julia> path = ccall(:getenv, Cstring, (Cstring,), "SHELL")
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

Note that the argument type tuple must be written as `(Cstring,)`, rather than `(Cstring)`. This is because `(Cstring)` is just the expression `Cstring` surrounded by parentheses, rather than a 1-tuple containing `Cstring`:

```
julia> (Cstring)
Cstring

julia> (Cstring,)
(Cstring,)
```

In practice, especially when providing reusable functionality, one generally wraps `ccall` uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function indicates them, propagating to the Julia caller as exceptions. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the `getenv` C library function is wrapped in the following Julia function, which is a simplified version of the actual definition from `env.jl`:

```
function getenv(var::AbstractString)
    val = ccall(:getenv, Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    return unsafe_string(val)
end
```

The C `getenv` function indicates an error by returning `NULL`, but other standard C functions indicate errors in various different ways, including by returning -1, 0, 1 and other special values. This wrapper throws an exception clearly indicating the problem if the caller tries to get a non-existent environment variable:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname. In this example, the networking library code is assumed to be in a shared library named "libc". In practice, this function is usually part of the C standard library, and so the "libc" portion should be omitted, but we wish to show here the usage of this syntax.

```
function gethostname()
    hostname = Vector{UInt8}(undef, 256) # MAXHOSTNAMELEN
    err = ccall((:gethostname, "libc"), Int32,
                (Ptr{UInt8}, Csize_t),
                hostname, sizeof(hostname))
    Base.systemerror("gethostname", err != 0)
    hostname[end] = 0 # ensure null-termination
    return unsafe_string(pointer(hostname))
end
```

This example first allocates an array of bytes, then calls the C library function `gethostname` to fill the array in with the hostname, takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a NUL-terminated C string. It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and filled in. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function. This is why we don't use the `Cstring` type here: as the array is uninitialized, it could contain NUL bytes. Converting to a `Cstring` as part of the `ccall` checks for contained NUL bytes and could therefore throw a conversion error.

30.1 Creating C-Compatible Julia Function Pointers

It is possible to pass Julia functions to native C functions that accept function pointer arguments. For example, to match C prototypes of the form:

```
typedef returntype (*functiontype)(argumenttype, ...)
```

The macro `@cfunction` generates the C-compatible function pointer for a call to a Julia function. The arguments to `@cfunction` are:

1. A Julia function
2. The function's return type
3. A tuple of input types, corresponding to the function signature

   Note

   As with `ccall`, the return type and tuple of input types must be literal constants.

> Note
>
> Currently, only the platform-default C calling convention is supported. This means that `@cfunction`-generated pointers cannot be used in calls where WINAPI expects `stdcall` function on 32-bit Windows, but can be used on WIN64 (where `stdcall` is unified with the C calling convention).

A classic example is the standard C library `qsort` function, declared as:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void*, const void*));
```

The `base` argument is a pointer to an array of length `nmemb`, with elements of `size` bytes each. `compare` is a callback function which takes pointers to two elements `a` and `b` and returns an integer less/greater than zero if `a` should appear before/after `b` (or zero if any order is permitted).

Now, suppose that we have a 1d array `A` of values in Julia that we want to sort using the `qsort` function (rather than Julia's built-in `sort` function). Before we worry about calling `qsort` and passing arguments, we need to write a comparison function:

```
julia> function mycompare(a, b)::Cint
           return (a < b) ? -1 : ((a > b) ? +1 : 0)
       end
mycompare (generic function with 1 method)
```

$qsort$ expects a comparison function that return a C int, so we annotate the return type to be $Cint$.

In order to pass this function to C, we obtain its address using the macro `@cfunction`:

```
julia> mycompare_c = @cfunction(mycompare, Cint, (Ref{Cdouble}, Ref{Cdouble}));
```

`@cfunction` requires three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a literal tuple of the input argument types, in this case to sort an array of `Cdouble` (`Float64`) elements.

The final call to `qsort` looks like this:

```
julia> A = [1.3, -2.7, 4.4, 3.1]
4-element Array{Float64,1}:
  1.3
 -2.7
  4.4
```

```
  3.1

julia> ccall(:qsort, Cvoid, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Cvoid}),
             A, length(A), sizeof(eltype(A)), mycompare_c)

julia> A
4-element Array{Float64,1}:
 -2.7
  1.3
  3.1
  4.4
```

As can be seen, `A` is changed to the sorted array `[-2.7, 1.3, 3.1, 4.4]`. Note that Julia takes care of converting the array to a `Ptr{Cdouble}`), computing the size of the element type in bytes, and so on.

For fun, try inserting a `println("mycompare($a, $b)")` line into `mycompare`, which will allow you to see the comparisons that `qsort` is performing (and to verify that it is really calling the Julia function that you passed to it).

30.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file.[2]

Automatic Type Conversion

Julia automatically inserts calls to the `Base.cconvert` function to convert each argument to the specified type. For example, the following call:

```
ccall((:foo, "libfoo"), Cvoid, (Int32, Float64), x, y)
```

will behave as if the following were written:

```
ccall((:foo, "libfoo"), Cvoid, (Int32, Float64),
      Base.unsafe_convert(Int32, Base.cconvert(Int32, x)),
      Base.unsafe_convert(Float64, Base.cconvert(Float64, y)))
```

`Base.cconvert` normally just calls `convert`, but can be defined to return an arbitrary new object more appropriate for passing to C. This should be used to perform all allocations of memory that will be accessed by the C code. For example, this is used to convert an `Array` of objects (e.g. strings) to an array of pointers.

`Base.unsafe_convert` handles conversion to `Ptr` types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences

First, let's review some relevant Julia type terminology:

Syntax / Keyword	Example	Description
`mutable struct`	`BitSet`	"Leaf Type" :: A group of related data that includes a type-tag, is managed by the Julia GC, and is defined by object-identity. The type parameters of a leaf type must be fully defined (no `TypeVars` are allowed) in order for the instance to be constructed.
`abstract type`	`Any`, `AbstractArray{T, N}`, `Complex{T}`	"Super Type" :: A super-type (not a leaf-type) that cannot be instantiated, but can be used to describe a group of types.
`T{A}`	`Vector{Int}`	"Type Parameter" :: A specialization of a type (typically used for dispatch or storage optimization).
		"TypeVar" :: The `T` in the type parameter declaration is referred to as a TypeVar (short for type variable).
`primitive type`	`Int`, `Float64`	"Primitive Type" :: A type with no fields, but a size. It is stored and defined by-value.
`struct`	`Pair{Int, Int}`	"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	`ComplexF64` (`isbits`)	"Is-Bits" :: A `primitive type`, or a `struct` type where all fields are other `isbits` types. It is defined by-value, and is stored without a type-tag.
`struct ...; end`	`nothing`	"Singleton" :: a Leaf Type or Struct with no fields.
`(...)` or `tuple(...)`	`(1, 2, 3)`	"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Bits Types

There are several special types to be aware of, as no other type can be defined to behave the same:

- `Float32`

  Exactly corresponds to the `float` type in C (or `REAL*4` in Fortran).

- `Float64`

  Exactly corresponds to the `double` type in C (or `REAL*8` in Fortran).

- `ComplexF32`

  Exactly corresponds to the `complex float` type in C (or `COMPLEX*8` in Fortran).

- `ComplexF64`

  Exactly corresponds to the `complex double` type in C (or `COMPLEX*16` in Fortran).

- `Signed`

  Exactly corresponds to the `signed` type annotation in C (or any `INTEGER` type in Fortran). Any Julia type that is not a subtype of `Signed` is assumed to be unsigned.

- `Ref{T}`

  Behaves like a `Ptr{T}` that can manage its memory via the Julia GC.

- `Array{T,N}`

  When an array is passed to C as a `Ptr{T}` argument, it is not reinterpret-cast: Julia requires that the element type of the array matches `T`, and the address of the first element is passed.

  Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `trunc(Int32, a)`.

  To pass an array `A` as a pointer of a different type without converting the data beforehand (for example, to pass a `Float64` array to a function that operates on uninterpreted bytes), you can declare the argument as `Ptr{Cvoid}`.

  If an array of eltype `Ptr{T}` is passed as a `Ptr{Ptr{T}}` argument, `Base.cconvert` will attempt to first make a null-terminated copy of the array with each element replaced by its `Base.cconvert` version. This allows, for example, passing an `argv` pointer array of type `Vector{String}` to an argument of type `Ptr{Ptr{Cchar}}`.

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help for writing portable code (and remembering that an `int` in C is not the same as an `Int` in Julia).

System Independent Types

The `Cstring` type is essentially a synonym for `Ptr{UInt8}`, except the conversion to `Cstring` throws an error if the Julia string contains any embedded NUL characters (which would cause the string to be silently truncated if the C routine treats NUL as the terminator). If you are passing a `char*` to a C routine that does not assume NUL termination

(e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain NUL and want to skip the check, you can use `Ptr{UInt8}` as the argument type. `Cstring` can also be used as the `ccall` return type, but in that case it obviously does not introduce any extra checks and is only meant to improve readability of the call.

System Dependent Types

> Note
>
> When calling Fortran, all inputs must be passed by pointers to heap- or stack-allocated values, so all type correspondences above should contain an additional `Ptr{..}` or `Ref{..}` wrapper around their type specification.

> Warning
>
> For string arguments (`char*`) the Julia type should be `Cstring` (if NUL- terminated data is expected) or either `Ptr{Cchar}` or `Ptr{UInt8}` otherwise (these two pointer types have the same effect), as described above, not `String`. Similarly, for array arguments (`T[]` or `T*`), the Julia type should again be `Ptr{T}`, not `Vector{T}`.

> Warning
>
> Julia's `Char` type is 32 bits, which is not the same as the wide character type (`wchar_t` or `wint_t`) on all platforms.

> Warning
>
> A return type of `Union{}` means the function will not return i.e. C++11 `[[noreturn]]` or C11 `_Noreturn` (e.g. `jl_throw` or `longjmp`). Do not use this for functions that return no value (`void`) but do return, use `Cvoid` instead.

> Note
>
> For `wchar_t*` arguments, the Julia type should be `Cwstring` (if the C routine expects a NUL-terminated string) or `Ptr{Cwchar_t}` otherwise. Note also that UTF-8 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy (but using the `Cwstring` type will cause an error to be thrown if the string itself contains NUL characters).

> Note
>
> C functions that take an argument of the type `char**` can be called by using a `Ptr{Ptr{UInt8}}` type within Julia. For example, C functions of the form:

```
int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{UInt8}}), length(argv), argv)
```

Note

For Fortran functions taking variable length strings of type `character(len=*)` the string lengths are provided as hidden arguments. Type and position of these arguments in the list are compiler specific, where compiler vendors usually default to using `Csize_t` as type and append the hidden arguments at the end of the argument list. While this behaviour is fixed for some compilers (GNU), others optionally permit placing hidden arguments directly after the character argument (Intel,PGI). For example, Fortran subroutines of the form

```
subroutine test(str1, str2)
character(len=*) :: str1,str2
```

can be called via the following Julia code, where the lengths are appended

```
str1 = "foo"
str2 = "bar"
ccall(:test, Void, (Ptr{UInt8}, Ptr{UInt8}, Csize_t, Csize_t),
                    str1, str2, sizeof(str1), sizeof(str2))
```

Warning

Fortran compilers may also add other hidden arguments for pointers, assumed-shape (`:`) and assumed-size (`*`) arrays. Such behaviour can be avoided by using `ISO_C_BINDING` and including `bind(c)` in the definition of the subroutine, which is strongly recommended for interoperable code. In this case there will be no hidden arguments, at the cost of some language features (e.g. only `character(len=1)` will be permitted to pass strings).

Note

A C function declared to return `Cvoid` will return the value `nothing` in Julia.

Struct Type Correspondences

Composite types, aka `struct` in C or `TYPE` in Fortran90 (or `STRUCTURE` / `RECORD` in some variants of F77), can be mirrored in Julia by creating a `struct` definition with the same field layout.

When used recursively, `isbits` types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead, declare an `isbits` struct type and use that instead. Unnamed structs are not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get an approximation of a `union` if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with `NTuple`. For example, the struct in C notation written as

```
struct B {
    int A[3];
};

b_a_2 = B.A[2];
```

can be written in Julia as

```
struct B
    A::NTuple{3, Cint}
end

b_a_2 = B.A[3]  # note the difference in indexing (1-based in Julia, 0-based in C)
```

Arrays of unknown size (C99-compliant variable length structs specified by `[]` or `[0]`) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

```
struct String {
    int strlen;
    char data[];
};
```

In Julia, we can access the parts independently to make a copy of that string:

```
str = from_c::Ptr{Cvoid}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

Type Parameters

The type arguments to `ccall` and `@cfunction` are evaluated statically, when the method containing the usage is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the intended C ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the call signature, as long as they don't affect the layout of the type. For example, `f(x::T) where {T} = ccall(:valid, Ptr{T}, (Ptr{T},), x)` is valid, since `Ptr` is always a word-size primitive type. But, `g(x::T) where {T} = ccall(:notvalid, T, (T,), x)` is not valid, since the type layout of `T` is not known statically.

SIMD Values

Note: This feature is currently implemented on 64-bit x86 and AArch64 platforms only.

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of `VecElement` that naturally maps to the SIMD type. Specifically:

- The tuple must be the same size as the SIMD type. For example, a tuple representing an `__m128` on x86 must have a size of 16 bytes.
- The element type of the tuple must be an instance of `VecElement{T}` where `T` is a primitive type that is 1, 2, 4 or 8 bytes.

For instance, consider this C routine that uses AVX intrinsics:

```
#include <immintrin.h>

__m256 dist( __m256 a, __m256 b ) {
    return _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(a, a),
                                        _mm256_mul_ps(b, b)));
}
```

The following Julia code calls `dist` using `ccall`:

```
const m256 = NTuple{8, VecElement{Float32}}

a = m256(ntuple(i -> VecElement(sin(Float32(i))), 8))
b = m256(ntuple(i -> VecElement(cos(Float32(i))), 8))
```

```
function call_dist(a::m256, b::m256)
    ccall((:dist, "libdist"), m256, (m256, m256), a, b)
end

println(call_dist(a,b))
```

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

Memory Ownership

malloc/free

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program. Do not try to free an object received from a C library with `Libc.free` in Julia, as this may result in the `free` function being called via the wrong library and cause the process to abort. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

When to use T, Ptr{T} and Ref{T}

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type `T` inside the `ccall`, as they are passed by value. For C code accepting pointers, `Ref{T}` should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to `Base.cconvert`. In contrast, pointers returned by the C function called should be declared to be of output type `Ptr{T}`, reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type `Ptr{T}` within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type `Ref{T}`, as Fortran passes all variables by pointers to memory locations. The return type should either be `Cvoid` for Fortran subroutines, or a `T` for Fortran functions returning the type `T`.

30.3 Mapping C Functions to Julia

`ccall` / `@cfunction` argument translation guide

For translating a C argument list to Julia:

- `T`, where `T` is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents

  - `T`, where `T` is an equivalent Julia Bits Type (per the table above)
  - if `T` is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (passed by value)
- `struct T` (including typedef to a struct)
  - `T`, where `T` is a Julia leaf type
  - argument value will be copied (passed by value)
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Cvoid}`, if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
- `jl_value_t**`
  - `Ref{Any}`
  - argument value must be a valid Julia object (or `C_NULL`)
- `T*`
  - `Ref{T}`, where `T` is the Julia type corresponding to `T`
  - argument value will be copied if it is an `isbits` type otherwise, the value must be a valid Julia object
- `T (*)(...)` (e.g. a pointer to a function)
  - `Ptr{Cvoid}` (you may need to use `@cfunction` explicitly to create this pointer)
- `...` (e.g. a vararg)
  - `T...`, where `T` is the Julia type
  - currently unsupported by `@cfunction`
- `va_arg`
  - not supported by `ccall` or `@cfunction`

`ccall` / `@cfunction` return type translation guide

For translating a C return type to Julia:

- `void`
  - `Cvoid` (this will return the singleton instance `nothing::Cvoid`)
- `T`, where `T` is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - `T`, where `T` is an equivalent Julia Bits Type (per the table above)
  - if `T` is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (returned by-value)
- `struct T` (including typedef to a struct)
  - `T`, where `T` is a Julia Leaf Type
  - argument value will be copied (returned by-value)
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Cvoid}`, if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
- `jl_value_t**`
  - `Ptr{Any}` (`Ref{Any}` is invalid as a return type)
  - argument value must be a valid Julia object (or `C_NULL`)
- `T*`
  - If the memory is already owned by Julia, or is an `isbits` type, and is known to be non-null:
    - `Ref{T}`, where `T` is the Julia type corresponding to `T`

        * a return type of `Ref{Any}` is invalid, it should either be `Any` (corresponding to `jl_value_t*`) or `Ptr{Any}` (corresponding to `jl_value_t**`)
        * C MUST NOT modify the memory returned via `Ref{T}` if `T` is an `isbits` type
    - If the memory is owned by C:
        * `Ptr{T}`, where `T` is the Julia type corresponding to `T`
- `T (*)(...)` (e.g. a pointer to a function)
    - `Ptr{Cvoid}` (you may need to use `@cfunction` explicitly to create this pointer)

Passing Pointers for Modifying Inputs

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a `ccall`, you need to first encapsulate the value inside a `Ref{T}` of the appropriate type. When you pass this `Ref` object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
ccall(:foo, Cvoid, (Ref{Cint}, Ref{Cfloat}), width, range)
```

Upon return, the contents of `width` and `range` can be retrieved (if they were changed by `foo`) by `width[]` and `range[]`; that is, they act like zero-dimensional arrays.

30.4 C Wrapper Examples

Let's start with a simple example of a C wrapper that returns a `Ptr` type:

```
mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = ccall(
        (:gsl_permutation_alloc, :libgsl), # name of C function and library
        Ptr{gsl_permutation},              # output type
        (Csize_t,),                        # tuple of input types
        n                                  # name of Julia variable to pass in
```

```
    )
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

The GNU Scientific Library (here assumed to be accessible through `:libgsl`) defines an opaque pointer, `gsl_permutation *`, as the return type of the C function `gsl_permutation_alloc`. As user code never has to look inside the `gsl_permutation` struct, the corresponding Julia wrapper simply needs a new type declaration, `gsl_permutation`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `Ptr` type. The return type of the `ccall` is declared as `Ptr{gsl_permutation}`, since the memory allocated and pointed to by `output_ptr` is controlled by C.

The input `n` is passed by value, and so the function's input signature is simply declared as `(Csize_t,)` without any `Ref` or `Ptr` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature would instead be `(Ref{Csize_t},)`, since Fortran variables are passed by pointers.) Furthermore, `n` can be any type that is convertible to a `Csize_t` integer; the `ccall` implicitly calls `Base.cconvert(Csize_t, n)`.

Here is a second example wrapping the corresponding destructor:

```
# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
function permutation_free(p::Ref{gsl_permutation})
    ccall(
        (:gsl_permutation_free, :libgsl), # name of C function and library
        Cvoid,                            # output type
        (Ref{gsl_permutation},),          # tuple of input types
        p                                 # name of Julia variable to pass in
    )
end
```

Here, the input `p` is declared to be of type `Ref{gsl_permutation}`, meaning that the memory that `p` points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type `Ptr{gsl_permutation}`, but it is convertible using `Base.cconvert` and therefore

Now if you look closely enough at this example, you may notice that it is incorrect, given our explanation above of preferred declaration types. Do you see it? The function we are calling is going to free the memory. This type of operation cannot be given a Julia object (it will crash or cause memory corruption). Therefore, it may be preferable to declare the `p` type as `Ptr{gsl_permutation }`, to make it harder for the user to mistakenly pass another sort of object there than one obtained via `gsl_permutation_alloc`.

If the C wrapper never expects the user to pass pointers to memory managed by Julia, then using `p::Ptr{gsl_permutation}` for the method signature of the wrapper and similarly in the `ccall` is also acceptable.

Here is a third example passing Julia arrays:

```
# The corresponding C signature is
#    int gsl_sf_bessel_Jn_array (int nmin, int nmax, double x,
#                                double result_array[])
function sf_bessel_Jn_array(nmin::Integer, nmax::Integer, x::Real)
    if nmax < nmin
        throw(DomainError())
    end
    result_array = Vector{Cdouble}(undef, nmax - nmin + 1)
    errorcode = ccall(
        (:gsl_sf_bessel_Jn_array, :libgsl), # name of C function and library
        Cint,                               # output type
        (Cint, Cint, Cdouble, Ref{Cdouble}),# tuple of input types
        nmin, nmax, x, result_array         # names of Julia variables to pass in
    )
    if errorcode != 0
        error("GSL error code $errorcode")
    end
    return result_array
end
```

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `result_array`. This variable is declared as a `Ref{Cdouble}`, since its memory is allocated and managed by Julia. The implicit call to `Base.cconvert(Ref{Cdouble}, result_array)` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

30.5 Fortran Wrapper Example

The following example utilizes ccall to call a function in a common Fortran library (libBLAS) to computes a dot product. Notice that the argument mapping is a bit different here than above, as we need to map from Julia to Fortran. On every argument type, we specify `Ref` or `Ptr`. This mangling convention may be specific to your fortran compiler and operating system, and is likely undocumented. However, wrapping each in a `Ref` (or `Ptr`, where equivalent) is a frequent requirement of Fortran compiler implementations:

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
```

```
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                    Float64,
                    (Ref{Int32}, Ptr{Float64}, Ref{Int32}, Ptr{Float64}, Ref{Int32}),
                    n, DX, incx, DY, incy)
    return product
end
```

30.6 Garbage Collection Safety

When passing data to a `ccall`, it is best to avoid using the `pointer` function. Instead define a convert method and pass the variables directly to the `ccall`. `ccall` automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the `ccall` returns, you must arrange that the object remains visible to the garbage collector. The suggested way to handle this is to make a global variable of type `Array{Ref,1}` to hold these values, until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you are done with using the pointer. Many methods in Julia such as `unsafe_load` and `String` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `unsafe_wrap` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if `a` contained a reference to `b` and both `a` and `b` are due for garbage collection, there is no guarantee that `b` would be finalized after `a`. If proper finalization of `a` depends on `b` being valid, it must be handled in other ways.

30.7 Non-constant Function Specifications

A `(name, library)` function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `eval` as follows:

```
@eval ccall(($(string("a", "b")), "lib"), ...
```

This expression constructs a name using `string`, then substitutes this name into a new `ccall` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `$`). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions. A similar example can be constructed for `@cfunction`.

However, doing this will also be very slow and leak memory, so you should usually avoid this and instead keep reading. The next section discusses how to use indirect calls to efficiently accomplish a similar effect.

30.8 Indirect Calls

The first argument to `ccall` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when the first `ccall` argument contains references to non-constants, such as local variables, function arguments, or non-constant globals.

For example, you might look up the function via `dlsym`, then cache it in a shared reference for that session. For example:

```
macro dlsym(func, lib)
    z = Ref{Ptr{Cvoid}}(C_NULL)
    quote
        let zlocal = $z[]
            if zlocal == C_NULL
                zlocal = dlsym($(esc(lib))::Ptr{Cvoid}, $(esc(func)))::Ptr{Cvoid}
                $z[] = $zlocal
            end
            zlocal
        end
    end
end

mylibvar = Libdl.dlopen("mylib")
ccall(@dlsym("myfunc", mylibvar), Cvoid, ())
```

30.9 Closure cfunctions

The first argument to `@cfunction` can be marked with a `$`, in which case the return value will instead be a `struct CFunction` which closes over the argument. You must ensure that this return object is kept alive until all uses of it are done. The contents and code at the cfunction pointer will be erased via a `finalizer` when this reference is dropped and atexit. This is not usually needed, since this functionality is not present in C, but can be useful for dealing with ill-designed APIs which don't provide a separate closure environment parameter.

```
function qsort(a::Vector{T}, cmp) where T
    isbits(T) || throw(ArgumentError("this method can only qsort isbits arrays"))
    callback = @cfunction $cmp Cint (Ref{T}, Ref{T})
```

```
    # Here, `callback` isa Base.CFunction, which will be converted to Ptr{Cvoid}
    # (and protected against finalization) by the ccall
    ccall(:qsort, Cvoid, (Ptr{T}, Csize_t, Csize_t, Ptr{Cvoid}),
        a, length(a), Base.elsize(a), callback)
    # We could instead use:
    #    GC.@preserve callback begin
    #        use(Base.unsafe_convert(Ptr{Cvoid}, callback))
    #    end
    # if we needed to use it outside of a `ccall`
    return a
end
```

30.10 Closing a Library

It is sometimes useful to close (unload) a library so that it can be reloaded. For instance, when developing C code for use with Julia, one may need to compile, call the C code from Julia, then close the library, make an edit, recompile, and load in the new changes. One can either restart Julia or use the `Libdl` functions to manage the library explicitly, such as:

```
lib = Libdl.dlopen("./my_lib.so") # Open the library explicitly.
sym = Libdl.dlsym(lib, :my_fcn)   # Get a symbol for the function to call.
ccall(sym, ...) # Use the pointer `sym` instead of the (symbol, library) tuple (remaining arguments are the
same).  Libdl.dlclose(lib) # Close the library explicitly.
```

Note that when using `ccall` with the tuple input (e.g., `ccall((:my_fcn, "./my_lib.so"), ...)`), the library is opened implicitly and it may not be explicitly closed.

30.11 Calling Convention

The second argument to ccall can optionally be a calling convention specifier (immediately preceding return type). Without any specifier, the platform-default C calling convention is used. Other supported conventions are: `stdcall`, `cdecl`, `fastcall`, and `thiscall` (no-op on 64-bit Windows). For example (from `base/libc.jl`) we see the same `gethostname`ccall as above, but with the correct signature for Windows:

```
hn = Vector{UInt8}(undef, 256)
err = ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))
```

For more information, please see the LLVM Language Reference.

There is one additional special calling convention `llvmcall`, which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for CUDA, we need to be able to read the thread index:

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmcall, Int32, ())
```

As with any `ccall`, it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by `Core.Intrinsics`.

30.12 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store!`.

> Note
>
> This `errno` symbol may not be found in a library named "libc", as this is an implementation detail of your system compiler. Typically standard library symbols should be accessed just by name, allowing the compiler to fill in the correct one. Also, however, the `errno` symbol shown in this example is special in most compilers, and so the value seen here is probably not what you expect or want. Compiling the equivalent code in C on any multi-threaded-capable system would typically actually call a different function (via macro preprocessor overloading), and may give a different result than the legacy value printed here.

30.13 Accessing Data through a Pointer

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a `Ptr{T}`, the contents of type `T` can generally be copied from the referenced memory into a Julia object using `unsafe_load(ptr, [index])`. The index argument is optional (default is 1), and follows the Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of `getindex` and `setindex!` (e.g. `[]` access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If `T` is `Any`, then the memory is assumed to contain a reference to a Julia object (a `jl_value_t*`), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. (Julia values `v` can be converted to `jl_value_t*` pointers, as `Ptr{Cvoid}`, by calling `pointer_from_objref(v)`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `unsafe_store!(ptr, value, [index])`. Currently, this is only supported for primitive types or other pointer-free (`isbits`) immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is a plain-data array (primitive type or immutable struct), the function `unsafe_wrap(Array, ptr,dims, own = false)` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the `own` parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr` type in Julia (e.g. using `+`) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr` in Julia always moves the pointer by some number of bytes, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

30.14 Thread-safety

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only schedule (via Julia's event loop) the execution of your "real" callback. To do this, create an `AsyncCondition` object and `wait` on it:

```
cond = Base.AsyncCondition()
wait(cond)
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cond.handle` as the argument, taking care to avoid any allocations or other interactions with the Julia runtime.

Note that events may be coalesced, so multiple calls to `uv_async_send` may result in a single wakeup notification to the condition.

30.15 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

30.16 C++

For direct C++ interfacing, see the Cxx package. For tools to create C++ bindings, see the CxxWrap package.

[1] Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. The point above is that the cost of actually doing foreign function call is about the same as doing a call in either native language.

[2] The Clang package can be used to auto-generate Julia code from a C header file.

C name	Fortran name	Standard Julia Alias	Julia Base Type
`unsigned char`	`CHARACTER`	`Cuchar`	`UInt8`
`bool` (only in C++)		`Cuchar`	`UInt8`
`short`	`INTEGER*2`, `LOGICAL*2`	`Cshort`	`Int16`
`unsigned short`		`Cushort`	`UInt16`
`int`, `BOOL` (C, typical)	`INTEGER*4`, `LOGICAL*4`	`Cint`	`Int32`
`unsigned int`		`Cuint`	`UInt32`
`long long`	`INTEGER*8`, `LOGICAL*8`	`Clonglong`	`Int64`
`unsigned long long`		`Culonglong`	`UInt64`
`intmax_t`		`Cintmax_t`	`Int64`
`uintmax_t`		`Cuintmax_t`	`UInt64`
`float`	`REAL*4i`	`Cfloat`	`Float32`
`double`	`REAL*8`	`Cdouble`	`Float64`
`complex float`	`COMPLEX*8`	`ComplexF32`	`Complex{Float32}`
`complex double`	`COMPLEX*16`	`ComplexF64`	`Complex{Float64}`
`ptrdiff_t`		`Cptrdiff_t`	`Int`
`ssize_t`		`Cssize_t`	`Int`
`size_t`		`Csize_t`	`UInt`
`void`			`Cvoid`
`void` and `[[noreturn]]` or `_Noreturn`			`Union{}`
`void*`			`Ptr{Cvoid}`
`T*` (where T represents an appropriately defined type)			`Ref{T}`
`char*` (or `char[]`, e.g. a string)	`CHARACTER*N`		`Cstring` if NUL-terminated, or `Ptr{UInt8}` if not
`char**` (or `*char[]`)			`Ptr{Ptr{UInt8}}`
`jl_value_t*` (any Julia Type)			`Any`
`jl_value_t**` (a reference to a Julia Type)			`Ref{Any}`
`va_arg`			Not supported
`...` (variadic function specification)			`T...` (where `T` is one of the above types, variadic functions of different argument types are not supported)

C name	Standard Julia Alias	Julia Base Type
`char`	`Cchar`	`Int8` (x86, x86_64), `UInt8` (powerpc, arm)
`long`	`Clong`	`Int` (UNIX), `Int32` (Windows)
`unsigned long`	`Culong`	`UInt` (UNIX), `UInt32` (Windows)
`wchar_t`	`Cwchar_t`	`Int32` (UNIX), `UInt16` (Windows)

Chapter 31

운영체제 변수 다루기

크로스 플랫폼(cross-plaform) 어플리케이션이나 라이브러리 설계는 운영체제 간의 작동방식을 다르게 할 필요가 생기기도 하다. `Sys.KERNEL` 변수는 이런 상황을 다룰 때 사용한다. `Sys` 모듈은 `Sys.KERNEL`변수를 더 편리하게 사용하기 위해 `isunix`, `islinux`, `isapple`, `isbsd`, `isfreebsd`, `iswindows`같은 함수를 지원한다:

```
if Sys.iswindows()
    windows_specific_thing(a)
end
```

`islinux`, `isapple`,`isfreebsd`은 `isunix`의 상호 배타적 부분 집합이다.

`@static` 매크로를 사용하여 특정 조건을 만족하지 않으면 함수를 실행하지 못하게 할 수 있다:

Simple blocks:

```
ccall((@static Sys.iswindows() ? :_fopen : :fopen), ...)
```

Complex blocks:

```
@static if Sys.islinux()
    linux_specific_thing(a)
else
    generic_thing(a)
end
```

조건문을 여럿 사용하는 경우 (`if`/`elseif`/`end`포함) `@static`을 각 단계마다 작성해야 한다 (괄호는 필요없지만 가독성을 위해 추천한다):

```
@static Sys.iswindows() ? :a : (@static Sys.isapple() ? :b : :c)
```

Chapter 32

Environment Variables

Julia can be configured with a number of environment variables, set either in the usual way for each operating system, or in a portable way from within Julia. Supposing that you want to set the environment variable `JULIA_EDITOR` to `vim`, you can type `ENV["JULIA_EDITOR"] = "vim"` (for instance, in the REPL) to make this change on a case by case basis, or add the same to the user configuration file `~/.julia/config/startup.jl` in the user's home directory to have a permanent effect. The current value of the same environment variable can be determined by evaluating `ENV["JULIA_EDITOR"]`.

The environment variables that Julia uses generally start with `JULIA`. If `InteractiveUtils.versioninfo` is called with the keyword `verbose=true`, then the output will list defined environment variables relevant for Julia, including those for which `JULIA` appears in the name.

> Note
>
> Some variables, such as `JULIA_NUM_THREADS` and `JULIA_PROJECT`, need to be set before Julia starts, therefore adding these to `~/.julia/config/startup.jl` is too late in the startup process. In Bash, environment variables can either be set manually by running, e.g., `export JULIA_NUM_THREADS=4` before starting Julia, or by adding the same command to `~/.bashrc` or `~/.bash_profile` to set the variable each time Bash is started.

32.1 File locations

`JULIA_BINDIR`

The absolute path of the directory containing the Julia executable, which sets the global variable `Sys.BINDIR`. If `$JULIA_BINDIR` is not set, then Julia determines the value `Sys.BINDIR` at run-time.

The executable itself is one of

```
$JULIA_BINDIR/julia
$JULIA_BINDIR/julia-debug
```

by default.

The global variable `Base.DATAROOTDIR` determines a relative path from `Sys.BINDIR` to the data directory associated with Julia. Then the path

```
$JULIA_BINDIR/$DATAROOTDIR/julia/base
```

determines the directory in which Julia initially searches for source files (via `Base.find_source_file()`).

Likewise, the global variable `Base.SYSCONFDIR` determines a relative path to the configuration file directory. Then Julia searches for a `startup.jl` file at

```
$JULIA_BINDIR/$SYSCONFDIR/julia/startup.jl
$JULIA_BINDIR/../etc/julia/startup.jl
```

by default (via `Base.load_julia_startup()`).

For example, a Linux installation with a Julia executable located at `/bin/julia`, a `DATAROOTDIR` of `../share`, and a `SYSCONFDIR` of `../etc` will have `JULIA_BINDIR` set to `/bin`, a source-file search path of

```
/share/julia/base
```

and a global configuration search path of

```
/etc/julia/startup.jl
```

JULIA_PROJECT

A directory path that indicates which project should be the initial active project. Setting this environment variable has the same effect as specifying the `--project` start-up option, but `--project` has higher precedence. If the variable is set to `@.` then Julia tries to find a project directory that contains `Project.toml` or `JuliaProject.toml` file from the current directory and its parents. See also the chapter on Code Loading.

> Note
>
> `JULIA_PROJECT` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

JULIA_LOAD_PATH

The `JULIA_LOAD_PATH` environment variable is used to populate the global Julia `LOAD_PATH` variable, which determines which packages can be loaded via `import` and `using` (see Code Loading).

Unlike the shell `PATH` variable, empty entries in `JULIA_LOAD_PATH` are expanded to the default value of `LOAD_PATH`, `["@", "@v#.#", "@stdlib"]` when populating `LOAD_PATH`. This allows easy appending, prepending, etc. of the load path value in shell scripts regardless of whether `JULIA_LOAD_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `LOAD_PATH` just do

```
export JULIA_LOAD_PATH="/foo/bar:$JULIA_LOAD_PATH"
```

If the `JULIA_LOAD_PATH` environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if `JULIA_LOAD_PATH` is not set, then it will be set to `/foo/bar:` which will expand to a `LOAD_PATH` value of `["/foo/bar", "@", "@v#.#", "@stdlib"]`. If `JULIA_LOAD_PATH` is set to the empty string, it expands to an empty `LOAD_PATH` array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty load path via the environment variable. If you want the default load path, either unset the environment variable or if it must have a value, set it to the string `:`.

JULIA_DEPOT_PATH

The `JULIA_DEPOT_PATH` environment variable is used to populate the global Julia `DEPOT_PATH` variable, which controls where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, configuration files, and the default location of the REPL's history file.

Unlike the shell `PATH` variable but similar to `JULIA_LOAD_PATH`, empty entries in `JULIA_DEPOT_PATH` are expanded to the default value of `DEPOT_PATH`. This allows easy appending, prepending, etc. of the depot path value in shell scripts regardless of whether `JULIA_DEPOT_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `DEPOT_PATH` just do

```
export JULIA_DEPOT_PATH="/foo/bar:$JULIA_DEPOT_PATH"
```

If the `JULIA_DEPOT_PATH` environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if `JULIA_DEPOT_PATH` is not set, then it will be set to `/foo/bar:` which will have the effect of prepending `/foo/bar` to the default depot path. If `JULIA_DEPOT_PATH` is set to the empty string, it expands to an empty `DEPOT_PATH` array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty depot path via the environment variable. If you want the default depot path, either unset the environment variable or if it must have a value, set it to the string `:`.

JULIA_HISTORY

The absolute path `REPL.find_hist_file()` of the REPL's history file. If `$JULIA_HISTORY` is not set, then `REPL.find_hist_file()` defaults to

```
$(DEPOT_PATH[1])/logs/repl_history.jl
```

JULIA_PKGRESOLVE_ACCURACY

A positive `Int` that determines how much time the max-sum subroutine `MaxSum.maxsum()` of the package dependency resolver will devote to attempting satisfying constraints before giving up: this value is by default `1`, and larger values correspond to larger amounts of time.

Suppose the value of `$JULIA_PKGRESOLVE_ACCURACY` is `n`. Then

- the number of pre-decimation iterations is `20*n`,
- the number of iterations between decimation steps is `10*n`, and
- at decimation steps, at most one in every `20*n` packages is decimated.

32.2 External applications

JULIA_SHELL

The absolute path of the shell with which Julia should execute external commands (via `Base.repl_cmd()`). Defaults to the environment variable `$SHELL`, and falls back to `/bin/sh` if `$SHELL` is unset.

> Note
>
> On Windows, this environment variable is ignored, and external commands are executed directly.

JULIA_EDITOR

The editor returned by `InteractiveUtils.editor()` and used in, e.g., `InteractiveUtils.edit`, referring to the command of the preferred editor, for instance `vim`.

`$JULIA_EDITOR` takes precedence over `$VISUAL`, which in turn takes precedence over `$EDITOR`. If none of these environment variables is set, then the editor is taken to be `open` on Windows and OS X, or `/etc/alternatives/editor` if it exists, or `emacs` otherwise.

32.3 Parallelization

JULIA_CPU_THREADS

Overrides the global variable `Base.Sys.CPU_THREADS`, the number of logical CPU cores available.

JULIA_WORKER_TIMEOUT

A `Float64` that sets the value of `Distributed.worker_timeout()` (default: `60.0`). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

JULIA_NUM_THREADS

An unsigned 64-bit integer (`uint64_t`) that sets the maximum number of threads available to Julia. If `$JULIA_NUM_THREADS` exceeds the number of available physical CPU cores, then the number of threads is set to the number of cores. If `$JULIA_NUM_THREADS` is not positive or is not set, or if the number of CPU cores cannot be determined through system calls, then the number of threads is set to `1`.

> Note
>
> `JULIA_NUM_THREADS` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

JULIA_THREAD_SLEEP_THRESHOLD

If set to a string that starts with the case-insensitive substring `"infinite"`, then spinning threads never sleep. Otherwise, `$JULIA_THREAD_SLEEP_THRESHOLD` is interpreted as an unsigned 64-bit integer (`uint64_t`) and gives, in nanoseconds, the amount of time after which spinning threads should sleep.

JULIA_EXCLUSIVE

If set to anything besides `0`, then Julia's thread policy is consistent with running on a dedicated machine: the master thread is on proc 0, and threads are affinitized. Otherwise, Julia lets the operating system handle thread policy.

32.4 REPL formatting

Environment variables that determine how REPL output should be formatted at the terminal. Generally, these variables should be set to ANSI terminal escape sequences. Julia provides a high-level interface with much of the same functionality; see the section on The Julia REPL.

`JULIA_ERROR_COLOR`

The formatting `Base.error_color()` (default: light red, `"\033[91m"`) that errors should have at the terminal.

`JULIA_WARN_COLOR`

The formatting `Base.warn_color()` (default: yellow, `"\033[93m"`) that warnings should have at the terminal.

`JULIA_INFO_COLOR`

The formatting `Base.info_color()` (default: cyan, `"\033[36m"`) that info should have at the terminal.

`JULIA_INPUT_COLOR`

The formatting `Base.input_color()` (default: normal, `"\033[0m"`) that input should have at the terminal.

`JULIA_ANSWER_COLOR`

The formatting `Base.answer_color()` (default: normal, `"\033[0m"`) that output should have at the terminal.

`JULIA_STACKFRAME_LINEINFO_COLOR`

The formatting `Base.stackframe_lineinfo_color()` (default: bold, `"\033[1m"`) that line info should have during a stack trace at the terminal.

`JULIA_STACKFRAME_FUNCTION_COLOR`

The formatting `Base.stackframe_function_color()` (default: bold, `"\033[1m"`) that function calls should have during a stack trace at the terminal.

32.5 Debugging and profiling

`JULIA_GC_ALLOC_POOL`, `JULIA_GC_ALLOC_OTHER`, `JULIA_GC_ALLOC_PRINT`

If set, these environment variables take strings that optionally start with the character `'r'`, followed by a string interpolation of a colon-separated list of three signed 64-bit integers (`int64_t`). This triple of integers `a:b:c` represents the arithmetic sequence `a`, `a + b`, `a + 2*b`, ... `c`.

- If it's the `n`th time that `jl_gc_pool_alloc()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_POOL`, then garbage collection is forced.
- If it's the `n`th time that `maybe_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_OTHER`, then garbage collection is forced.

- If it's the nth time that `jl_gc_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_PRINT`, then counts for the number of calls to `jl_gc_pool_alloc()` and `maybe_collect()` are printed.

If the value of the environment variable begins with the character `'r'`, then the interval between garbage collection events is randomized.

> Note
>
> These environment variables only have an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`JULIA_GC_NO_GENERATIONAL`

If set to anything besides `0`, then the Julia garbage collector never performs "quick sweeps" of memory.

> Note
>
> This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`JULIA_GC_WAIT_FOR_DEBUGGER`

If set to anything besides `0`, then the Julia garbage collector will wait for a debugger to attach instead of aborting whenever there's a critical error.

> Note
>
> This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to `1` in the build configuration).

`ENABLE_JITPROFILING`

If set to anything besides `0`, then the compiler will create and register an event listener for just-in-time (JIT) profiling.

> Note
>
> This environment variable only has an effect if Julia was compiled with JIT profiling support, using either
>
> - Intel's VTune™ Amplifier (`USE_INTEL_JITEVENTS` set to `1` in the build configuration), or
> - OProfile (`USE_OPROFILE_JITEVENTS` set to `1` in the build configuration).

JULIA_LLVM_ARGS

Arguments to be passed to the LLVM backend.

JULIA_DEBUG_LOADING

If set, then Julia prints detailed information about the cache in the loading process of `Base.require`.

Chapter 33

Embedding Julia

As we have seen in Calling C and Fortran Code, Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia function from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python or C#).

33.1 High-Level Embedding

Note: This section covers embedding Julia code in C on Unix-like operating systems. For doing this on Windows, please see the section following this.

We start with a simple C program that initializes Julia and calls some Julia code:

```
#include <julia.h>
JULIA_DEFINE_FAST_TLS() // only define this once, in an executable (not in a shared library) if you want fast code.

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
```

```
    */
    jl_atexit_hook(0);
    return 0;
}
```

In order to build this program you have to put the path to the Julia header into the include path and link against `libjulia`. For instance, when Julia is installed to `$JULIA_DIR`, one can compile the above test program `test.c` with `gcc` using:

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib test.c -ljulia $JULIA_DIR/lib/julia/libstdc++.so.6
```

Then if the environment variable `JULIA_BINDIR` is set to `$JULIA_DIR/bin`, the output `test` program can be executed.

Alternatively, look at the `embedding.c` program in the Julia source tree in the `test/embedding/` folder. The file `ui/repl.c` program is another simple example of how to set `jl_options` options while linking against `libjulia`.

The first thing that has to be done before calling any other Julia C function is to initialize Julia. This is done by calling `jl_init`, which tries to automatically determine Julia's install location. If you need to specify a custom location, or specify which system image to load, use `jl_init_with_image` instead.

The second statement in the test program evaluates a Julia statement using a call to `jl_eval_string`.

Before the program terminates, it is strongly recommended to call `jl_atexit_hook`. The above example program calls this before returning from `main`.

> Note
>
> Currently, dynamically linking with the `libjulia` shared library requires passing the `RTLD_GLOBAL` option.
> In Python, this looks like:
>
> ```
> >>> julia=CDLL('./libjulia.dylib',RTLD_GLOBAL)
> >>> julia.jl_init.argtypes = []
> >>> julia.jl_init()
> 250593296
> ```

> Note
>
> If the julia program needs to access symbols from the main executable, it may be necessary to add `-Wl,--export-dynamic` linker flag at compile time on Linux in addition to the ones generated by `julia-config.jl` described below. This is not necessary when compiling a shared library.

Using julia-config to automatically determine build parameters

The script `julia-config.jl` was created to aid in determining what build parameters are required by a program that uses embedded Julia. This script uses the build parameters and system configuration of the particular Julia distribution it is invoked by to export the necessary compiler flags for an embedding program to interact with that distribution. This script is located in the Julia shared data directory.

Example

```
#include <julia.h>


int main(int argc, char *argv[])
{
    jl_init();
    (void)jl_eval_string("println(sqrt(2.0))");
    jl_atexit_hook(0);
    return 0;
}
```

On the command line

A simple use of this script is from the command line. Assuming that `julia-config.jl` is located in `/usr/local/julia/share/julia`, it can be invoked on the command line directly and takes any combination of 3 flags:

```
/usr/local/julia/share/julia/julia-config.jl
Usage: julia-config [--cflags|--ldflags|--ldlibs]
```

If the above example source is saved in the file `embed_example.c`, then the following command will compile it into a running program on Linux and Windows (MSYS2 environment), or if on OS/X, then substitute `clang` for `gcc`.:

```
/usr/local/julia/share/julia/julia-config.jl --cflags --ldflags --ldlibs | xargs gcc embed_example.c
```

Use in Makefiles

But in general, embedding projects will be more complicated than the above, and so the following allows general makefile support as well — assuming GNU make because of the use of the shell macro expansions. Additionally, though many times `julia-config.jl` may be found in the directory `/usr/local`, this is not necessarily the case, but Julia can be used to locate `julia-config.jl` too, and the makefile can be used to take advantage of that. The above example is extended to use a Makefile:

```
JL_SHARE = $(shell julia -e 'print(joinpath(Sys.BINDIR, Base.DATAROOTDIR, "julia"))')
CFLAGS   += $(shell $(JL_SHARE)/julia-config.jl --cflags)
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
```

```
LDFLAGS  += $(shell $(JL_SHARE)/julia-config.jl --ldflags)
LDLIBS   += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)

all: embed_example
```

Now the build command is simply `make`.

33.2 High-Level Embedding on Windows with Visual Studio

If the `JULIA_DIR` environment variable hasn't been setup, add it using the System panel before starting Visual Studio. The `bin` folder under JULIA_DIR should be on the system PATH.

We start by opening Visual Studio and creating a new Console Application project. To the 'stdafx.h' header file, add the following lines at the end:

```
#define JULIA_ENABLE_THREADING
#include <julia.h>
```

Then, replace the main() function in the project with this code:

```
int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

The next step is to set up the project to find the Julia include files and the libraries. It's important to know whether the Julia installation is 32- or 64-bits. Remove any platform configuration that doesn't correspond to the Julia installation before proceeding.

Using the project Properties dialog, go to `C/C++` | `General` and add `$(JULIA_DIR)\include\julia\` to the Additional Include Directories property. Then, go to the `Linker` | `General` section and add `$(JULIA_DIR)\lib` to the Additional Library Directories property. Finally, under `Linker` | `Input`, add `libjulia.dll.a;libopenlibm.dll.a;` to the list of libraries.

At this point, the project should build and run.

33.3 Converting Types

Real applications will not just need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `Float64` in this way is called `boxing`, and extracting the stored primitive data is called `unboxing`. Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C looks as follows:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");

if (jl_typeis(ret, jl_float64_type)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
else {
    printf("ERROR: unexpected return type from sqrt(::Float64)\n");
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `jl_isa`, `jl_typeis`, or `jl_is_...` functions. By typing `typeof(sqrt(2.0))` into the Julia shell we can see that the return type is `Float64` (`double` in C). To convert the boxed Julia value into a C double the `jl_unbox_float64` function is used in the above code snippet.

Corresponding `jl_box_...` functions are used to convert the other way:

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

As we will see next, boxing is required to call Julia functions with specific arguments.

33.4 Calling Julia Functions

While `jl_eval_string` allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using `jl_call`:

```
jl_function_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);
```

In the first step, a handle to the Julia function `sqrt` is retrieved by calling `jl_get_function`. The first argument passed to `jl_get_function` is a pointer to the `Base` module in which `sqrt` is defined. Then, the double value is boxed using `jl_box_float64`. Finally, in the last step, the function is called using `jl_call1`. `jl_call0`, `jl_call2`, and `jl_call3` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `jl_call`:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)
```

Its second argument `args` is an array of `jl_value_t*` arguments and `nargs` is the number of arguments.

33.5 Memory Management

As we have seen, Julia objects are represented in C as pointers. This raises the question of who is responsible for freeing these objects.

Typically, Julia objects are freed by a garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.

The GC can only run when Julia objects are allocated. Calls like `jl_box_float64` perform allocation, and allocation might also happen at any point in running Julia code. However, it is generally safe to use pointers in between `jl_...` calls. But in order to make sure that values can survive `jl_...` calls, we have to tell Julia that we hold a reference to a Julia value. This can be done using the `JL_GC_PUSH` macros:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret);
// Do something with ret
JL_GC_POP();
```

The `JL_GC_POP` call releases the references established by the previous `JL_GC_PUSH`. Note that `JL_GC_PUSH` stores references on the C stack, so it must be exactly paired with a `JL_GC_POP` before the scope is exited. That is, before the function returns, or control flow otherwise leaves the block in which the `JL_GC_PUSH` was invoked.

Several Julia values can be pushed at once using the `JL_GC_PUSH2` , `JL_GC_PUSH3` , `JL_GC_PUSH4` , `JL_GC_PUSH5` , and `JL_GC_PUSH6` macros. To push an array of Julia values one can use the `JL_GC_PUSHARGS` macro, which can be used as follows:

```
jl_value_t **args;
JL_GC_PUSHARGS(args, 2); // args can now hold 2 `jl_value_t*` objects
```

```
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call jl_... functions)
JL_GC_POP();
```

Each scope must have only one call to `JL_GC_PUSH*`. Hence, if all variables cannot be pushed once by a single call to `JL_GC_PUSH*`, or if there are more than 6 variables to be pushed and using an array of arguments is not an option, then one can use inner blocks:

```
jl_value_t *ret1 = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret1);
jl_value_t *ret2 = 0;
{
    jl_function_t *func = jl_get_function(jl_base_module, "exp");
    ret2 = jl_call1(func, ret1);
    JL_GC_PUSH1(&ret2);
    // Do something with ret2.
    JL_GC_POP();    // This pops ret2.
}
JL_GC_POP();    // This pops ret1.
```

If it is required to hold the pointer to a variable between functions (or block scopes), then it is not possible to use `JL_GC_PUSH*`. In this case, it is necessary to create and keep a reference to the variable in the Julia global scope. One simple way to accomplish this is to use a global `IdDict` that will hold the references, avoiding deallocation by the GC. However, this method will only work properly with mutable types.

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");

...

// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Vector{Float64}`, which is mutable.
var = jl_eval_string("[sqrt(2.0); sqrt(4.0); sqrt(6.0)]");

// To protect `var`, add its reference to `refs`.
```

```
jl_call3(setindex, refs, var, var);
```

If the variable is immutable, then it needs to be wrapped in an equivalent mutable container or, preferably, in a `RefValue{Any}` before it is pushed to `IdDict`. In this approach, the container has to be created or filled in via C code using, for example, the function `jl_new_struct`. If the container is created by `jl_call*`, then you will need to reload the pointer to be used in C code.

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");
jl_datatype_t* reft = (jl_datatype_t*)jl_eval_string("Base.RefValue{Any}");

...

// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Float64`, which is immutable.
var = jl_eval_string("sqrt(2.0)");

// Protect `var` until we add its reference to `refs`.
JL_GC_PUSH1(&var);

// Wrap `var` in `RefValue{Any}` and push to `refs` to protect it.
jl_value_t* rvar = jl_new_struct(reft, var);
JL_GC_POP();

jl_call3(setindex, refs, rvar, rvar);
```

The GC can be allowed to deallocate a variable by removing the reference to it from `refs` using the function `delete!`, provided that no other reference to the variable is kept anywhere:

```
jl_function_t* delete = jl_get_function(jl_base_module, "delete!");
jl_call2(delete, refs, rvar);
```

As an alternative for very simple cases, it is possible to just create a global container of type `Vector{Any}` and fetch the elements from that when necessary, or even to create one global variable per pointer using

```
jl_set_global(jl_main_module, jl_symbol("var"), var);
```

Updating fields of GC-managed objects

The garbage collector operates under the assumption that it is aware of every old-generation object pointing to a young-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the `jl_gc_wb` (write barrier) function like so:

```
jl_value_t *parent = some_old_value, *child = some_young_value;
((some_specific_type*)parent)->field = child;
jl_gc_wb(parent, child);
```

It is in general impossible to predict which values will be old at runtime, so the write barrier must be inserted after all explicit stores. One notable exception is if the `parent` object was just allocated and garbage collection was not run since then. Remember that most `jl_...` functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. For example:

```
jl_array_t *some_array = ...; // e.g. a Vector{Any}
void **data = (void**)jl_array_data(some_array);
jl_value_t *some_value = ...;
data[0] = some_value;
jl_gc_wb(some_array, some_value);
```

Manipulating the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
`jl_gc_collect()`	Force a GC run
`jl_gc_enable(0)`	Disable the GC, return previous state as int
`jl_gc_enable(1)`	Enable the GC, return previous state as int
`jl_gc_is_enabled()`	Return current state as int

33.6 Working with Arrays

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype `jl_array_t*`. Basically, `jl_array_t` is a struct that contains:

- Information about the datatype
- A pointer to the data block

- Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing Float64 elements of length 10 is done by:

```
jl_value_t* array_type = jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call `free` on the data pointer when the array is no longer referenced.

In order to access the data of `x`, we can use `jl_array_data`:

```
double *xData = (double*)jl_array_data(x);
```

Now we can fill the array:

```
for(size_t i=0; i<jl_array_len(x); i++)
    xData[i] = i;
```

Now let us call a Julia function that performs an in-place operation on `x`:

```
jl_function_t *func = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

By printing the array, one can verify that the elements of `x` are now reversed.

Accessing Returned Arrays

If a Julia function returns an array, the return value of `jl_eval_string` and `jl_call` can be cast to a `jl_array_t*`:

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

Now the content of `y` can be accessed as before using `jl_array_data`. As always, be sure to keep a reference to the array while it is in use.

Multidimensional Arrays

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type(jl_float64_type, 2);
jl_array_t *x  = jl_alloc_array_2d(array_type, 10, 5);

// Get array pointer
double *p = (double*)jl_array_data(x);
// Get number of dimensions
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
    for(size_t j=0; j<size0; j++)
        p[j + size0*i] = i + j;
```

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_dim`) in order to read as idiomatic C code.

33.7 Exceptions

Julia code can throw exceptions. For example, consider:

```
jl_eval_string("this_function_does_not_exist()");
```

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into `libjulia` with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

Throwing Julia Exceptions

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

```
if (!jl_typeis(val, jl_float64_type)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

General exceptions can be raised using the functions:

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` takes a C string, and `jl_errorf` is called like `printf`:

```
jl_errorf("argument x = %d is too large", x);
```

where in this example `x` is assumed to be an integer.

Chapter 34

Code Loading

> Note
>
> This chapter covers the technical details of package loading. To install packages, use Pkg, Julia's built-in package manager, to add packages to your active environment. To use packages already in your active environment, write `import X` or `using X`, as described in the Modules documentation.

34.1 Definitions

Julia has two mechanisms for loading code:

1. Code inclusion: e.g. `include("source.jl")`. Inclusion allows you to split a single program across multiple source files. The expression `include("source.jl")` causes the contents of the file `source.jl` to be evaluated in the global scope of the module where the `include` call occurs. If `include("source.jl")` is called multiple times, `source.jl` is evaluated multiple times. The included path, `source.jl`, is interpreted relative to the file where the `include` call occurs. This makes it simple to relocate a subtree of source files. In the REPL, included paths are interpreted relative to the current working directory, `pwd()`.

2. Package loading: e.g. `import X` or `using X`. The import mechanism allows you to load a package—i.e. an independent, reusable collection of Julia code, wrapped in a module—and makes the resulting module available by the name `X` inside of the importing module. If the same `X` package is imported multiple times in the same Julia session, it is only loaded the first time—on subsequent imports, the importing module gets a reference to the same module. Note though, that `import X` can load different packages in different contexts: `X` can refer to one package named `X` in the main project but potentially to different packages also named `X` in each dependency. More on this below.

Code inclusion is quite straightforward and simple: it evaluates the given source file in the context of the caller. Package loading is built on top of code inclusion and serves a different purpose. The rest of this chapter focuses on the behavior and mechanics of package loading.

A package is a source tree with a standard layout providing functionality that can be reused by other Julia projects. A package is loaded by `import X` or `using X` statements. These statements also make the module named `X`—which results from loading the package code—available within the module where the import statement occurs. The meaning of `X` in `import X` is context-dependent: which `X` package is loaded depends on what code the statement occurs in. Thus, handling of `import X` happens in two stages: first, it determines what package is defined to be `X` in this context; second, it determines where that particular `X` package is found.

These questions are answered by searching through the project environments listed in `LOAD_PATH` for project files (`Project.toml` or `JuliaProject.toml`), manifest files (`Manifest.toml` or `JuliaManifest.toml`), or folders of source files.

34.2 Federation of packages

Most of the time, a package is uniquely identifiable simply from its name. However, sometimes a project might encounter a situation where it needs to use two different packages that share the same name. While you might be able fix this by renaming one of the packages, being forced to do so can be highly disruptive in a large, shared code base. Instead, Julia's code loading mechanism allows the same package name to refer to different packages in different components of an application.

Julia supports federated package management, which means that multiple independent parties can maintain both public and private packages and registries of packages, and that projects can depend on a mix of public and private packages from different registries. Packages from various registries are installed and managed using a common set of tools and workflows. The `Pkg` package manager that ships with Julia lets you install and manage your projects' dependencies. It assists in creating and manipulating project files (which describe what other projects that your project depends on), and manifest files (which snapshot exact versions of your project's complete dependency graph).

One consequence of federation is that there cannot be a central authority for package naming. Different entities may use the same name to refer to unrelated packages. This possibility is unavoidable since these entities do not coordinate and may not even know about each other. Because of the lack of a central naming authority, a single project may end up depending on different packages that have the same name. Julia's package loading mechanism does not require package names to be globally unique, even within the dependency graph of a single project. Instead, packages are identified by universally unique identifiers (UUIDs), which get assigned when each package is created. Usually you won't have to work directly with these somewhat cumbersome 128-bit identifiers since `Pkg` will take care of generating and tracking them for you. However, these UUIDs provide the definitive answer to the question of "what package does `X` refer to?"

Since the decentralized naming problem is somewhat abstract, it may help to walk through a concrete scenario to understand the issue. Suppose you're developing an application called `App`, which uses two packages: `Pub` and `Priv`. `Priv` is a private package that you created, whereas `Pub` is a public package that you use but don't control. When you created `Priv`, there was no public package by the name `Priv`. Subsequently, however, an unrelated package also named `Priv` has been published and become popular. In fact, the `Pub` package has started to use it. Therefore, when you next upgrade `Pub` to get the latest bug fixes and features, `App` will end up depending on two different packages named `Priv`—through no action of yours other than upgrading. `App` has a direct dependency on your private `Priv` package, and an indirect dependency, through `Pub`, on the new public `Priv` package. Since these two `Priv` packages are different but are both required for `App` to continue working correctly, the expression `import Priv` must refer to different `Priv` packages depending on whether it occurs in `App`'s code or in `Pub`'s code. To handle this, Julia's package loading mechanism distinguishes the two `Priv` packages by their UUID and picks the correct one based on its context (the module that called `import`). How this distinction works is determined by environments, as explained in the following sections.

34.3 Environments

An environment determines what `import X` and `using X` mean in various code contexts and what files these statements cause to be loaded. Julia understands two kinds of environments:

1. A project environment is a directory with a project file and an optional manifest file, and forms an explicit environement. The project file determines what the names and identities of the direct dependencies of a project are. The manifest file, if present, gives a complete dependency graph, including all direct and indirect dependencies, exact versions of each dependency, and sufficient information to locate and load the correct version.

2. A package directory is a directory containing the source trees of a set of packages as subdirectories, and forms an implicit environment. If `X` is a subdirectory of a package directory and `X/src/X.jl` exists, then the package `X` is available in the package directory environment and `X/src/X.jl` is the source file by which it is loaded.

These can be intermixed to create a stacked environment: an ordered set of project environments and package directories, overlaid to make a single composite environment. The precedence and visibility rules then combine to determine which packages are available and where they get loaded from. Julia's load path forms a stacked environment, for example.

These environment each serve a different purpose:

- Project environments provide reproducibility. By checking a project environment into version control—e.g. a git repository—along with the rest of the project's source code, you can reproduce the exact state of the project

and all of its dependencies. The manifest file, in particular, captures the exact version of every dependency, identified by a cryptographic hash of its source tree, which makes it possible for `Pkg` to retrieve the correct versions and be sure that you are running the exact code that was recorded for all dependencies.

- Package directories provide convenience when a full carefully-tracked project environment is unnecessary. They are useful when you want to put a set of packages somewhere and be able to directly use them, without needing to create a project environment for them.

- Stacked environments allow for adding tools to the primary environment. You can push an environment of development tools onto the end of the stack to make them available from the REPL and scripts, but not from inside packages.

At a high-level, each environment conceptually defines three maps: roots, graph and paths. When resolving the meaning of `import X`, the roots and graph maps are used to determine the identity of `X`, while the paths map is used to locate the source code of `X`. The specific roles of the three maps are:

- roots: `name::Symbol` ⨳ `uuid::UUID`

  An environment's roots map assigns package names to UUIDs for all the top-level dependencies that the environment makes available to the main project (i.e. the ones that can be loaded in `Main`). When Julia encounters `import X` in the main project, it looks up the identity of `X` as `roots[:X]`.

- graph: `context::UUID` ⨳ `name::Symbol` ⨳ `uuid::UUID`

  An environment's graph is a multilevel map which assigns, for each `context` UUID, a map from names to UUIDs, similar to the roots map but specific to that `context`. When Julia sees `import X` in the code of the package whose UUID is `context`, it looks up the identity of `X` as `graph[context][:X]`. In particular, this means that `import X` can refer to different packages depending on `context`.

- paths: `uuid::UUID` × `name::Symbol` ⨳ `path::String`

  The paths map assigns to each package UUID-name pair, the location of that package's entry-point source file. After the identity of `X` in `import X` has been resolved to a UUID via roots or graph (depending on whether it is loaded from the main project or a dependency), Julia determines what file to load to acquire `X` by looking up `paths[uuid,:X]` in the environment. Including this file should define a module named `X`. Once this package is loaded, any subsequent import resolving to the same `uuid` will create a new binding to the already-loaded package module.

Each kind of environment defines these three maps differently, as detailed in the following sections.

> Note
>
> For ease of understanding, the examples throughout this chapter show full data structures for roots, graph and paths. However, Julia's package loading code does not explicitly create these. Instead, it lazily computes only as much of each structure as it needs to load a given package.

Project environments

A project environment is determined by a directory containing a project file called `Project.toml`, and optionally a manifest file called `Manifest.toml`. These files may also be called `JuliaProject.toml` and `JuliaManifest.toml`, in which case `Project.toml` and `Manifest.toml` are ignored. This allows for coexistence with other tools that might consider files called `Project.toml` and `Manifest.toml` significant. For pure Julia projects, however, the names `Project.toml` and `Manifest.toml` are preferred.

The roots, graph and paths maps of a project environment are defined as follows:

The roots map of the environment is determined by the contents of the project file, specifically, its top-level `name` and `uuid` entries and its `[deps]` section (all optional). Consider the following example project file for the hypothetical application, `App`, as described earlier:

```
name = "App"
uuid = "8f986787-14fe-4607-ba5d-fbff2944afa9"

[deps]
Priv = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
Pub  = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
```

This project file implies the following roots map, if it was represented by a Julia dictionary:

```
roots = Dict(
    :App  => UUID("8f986787-14fe-4607-ba5d-fbff2944afa9"),
    :Priv => UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"),
    :Pub  => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
)
```

Given this roots map, in `App`'s code the statement `import Priv` will cause Julia to look up `roots[:Priv]`, which yields `ba13f791-ae1d-465a-978b-69c3ad90f72b`, the UUID of the `Priv` package that is to be loaded in that context. This UUID identifies which `Priv` package to load and use when the main application evaluates `import Priv`.

The dependency graph of a project environment is determined by the contents of the manifest file, if present. If there is no manifest file, graph is empty. A manifest file contains a stanza for each of a project's direct or indirect

dependencies. For each dependency, the file lists the package's UUID and a source tree hash or an explicit path to the source code. Consider the following example manifest file for `App`:

```
[[Priv]] # the private one
deps = ["Pub", "Zebra"]
uuid = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
path = "deps/Priv"

[[Priv]] # the public one
uuid = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
git-tree-sha1 = "1bf63d3be994fe83456a03b874b409cfd59a6373"
version = "0.1.5"

[[Pub]]
uuid = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
git-tree-sha1 = "9ebd50e2b0dd1e110e842df3b433cb5869b0dd38"
version = "2.1.4"

  [Pub.deps]
  Priv = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
  Zebra = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"

[[Zebra]]
uuid = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"
git-tree-sha1 = "e808e36a5d7173974b90a15a353b564f3494092f"
version = "3.4.2"
```

This manifest file describes a possible complete dependency graph for the `App` project:

- There are two different packages named `Priv` that the application uses. It uses a private package, which is a root dependency, and a public one, which is an indirect dependency through `Pub`. These are differentiated by their distinct UUIDs, and they have different deps:
  - The private `Priv` depends on the `Pub` and `Zebra` packages.
  - The public `Priv` has no dependencies.
- The application also depends on the `Pub` package, which in turn depends on the public `Priv` and the same `Zebra` package that the private `Priv` package depends on.

This dependency graph represented as a dictionary, looks like this:

```
graph = Dict(
    # Priv – the private one:
    UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b") => Dict(
        :Pub   => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Priv – the public one:
    UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c") => Dict(),
    # Pub:
    UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1") => Dict(
        :Priv  => UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Zebra:
    UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62") => Dict(),
)
```

Given this dependency `graph`, when Julia sees `import Priv` in the `Pub` package—which has UUID `c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1`—it looks up:

```
graph[UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1")][:Priv]
```

and gets `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`, which indicates that in the context of the `Pub` package, `import Priv` refers to the public `Priv` package, rather than the private one which the app depends on directly. This is how the name `Priv` can refer to different packages in the main project than it does in one of its package's dependencies, which allows for duplicate names in the package ecosystem.

What happens if `import Zebra` is evaluated in the main `App` code base? Since `Zebra` does not appear in the project file, the import will fail even though `Zebra` does appear in the manifest file. Moreover, if `import Zebra` occurs in the public `Priv` package—the one with UUID `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`—then that would also fail since that `Priv` package has no declared dependencies in the manifest file and therefore cannot load any packages. The `Zebra` package can only be loaded by packages for which it appear as an explicit dependency in the manifest file: the `Pub` package and one of the `Priv` packages.

The paths map of a project environment is extracted from the manifest file. The path of a package `uuid` named `X` is determined by these rules (in order):

1. If the project file in the directory matches `uuid` and name `X`, then either:

- It has a toplevel `path` entry, then `uuid` will be mapped to that path, interpreted relative to the directory containing the project file.
- Otherwise, `uuid` is mapped to `src/X.jl` relative to the directory containing the project file.

2. If the above is not the case and the project file has a corresponding manifest file and the manifest contains a stanza matching `uuid` then:

- If it has a `path` entry, use that path (relative to the directory containing the manifest file).
- If it has a `git-tree-sha1` entry, compute a deterministic hash function of `uuid` and `git-tree-sha1`—call it `slug`—and look for a directory named `packages/X/$slug` in each directory in the Julia `DEPOT_PATH` global array. Use the first such directory that exists.

If any of these result in success, the path to the source code entry point will be either that result, the relative path from that result plus `src/X.jl`; otherwise, there is no path mapping for `uuid`. When loading `X`, if no source code path is found, the lookup will fail, and the user may be prompted to install the appropriate package version or to take other corrective action (e.g. declaring `X` as a dependency).

In the example manifest file above, to find the path of the first `Priv` package—the one with UUID `ba13f791-ae1d-465a-978b-69c3ad90f72b`—Julia looks for its stanza in the manifest file, sees that it has a `path` entry, looks at `deps/Priv` relative to the `App` project directory—let's suppose the `App` code lives in `/home/me/projects/App`—sees that `/home/me/projects/App/deps/Priv` exists and therefore loads `Priv` from there.

If, on the other hand, Julia was loading the other `Priv` package—the one with UUID `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`—it finds its stanza in the manifest, see that it does not have a `path` entry, but that it does have a `git-tree-sha1` entry. It then computes the `slug` for this UUID/SHA-1 pair, which is `HDkrT` (the exact details of this computation aren't important, but it is consistent and deterministic). This means that the path to this `Priv` package will be `packages/Priv/HDkrT/src/Priv.jl` in one of the package depots. Suppose the contents of `DEPOT_PATH` is `["/home/me/.julia", "/usr/local/julia"]`, then Julia will look at the following paths to see if they exist:

1. `/home/me/.julia/packages/Priv/HDkrT`
2. `/usr/local/julia/packages/Priv/HDkrT`

Julia uses the first of these that exists to try to load the public `Priv` package from the file `packages/Priv/HDkrT/src/Priv.jl` in the depot where it was found.

Here is a representation of a possible paths map for our example `App` project environment, as provided in the Manifest given above for the dependency graph, after searching the local file system:

```
paths = Dict(
    # Priv — the private one:
    (UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"), :Priv) =>
        # relative entry-point inside `App` repo:
        "/home/me/projects/App/deps/Priv/src/Priv.jl",
    # Priv — the public one:
    (UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"), :Priv) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Priv/HDkr/src/Priv.jl",
    # Pub:
    (UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"), :Pub) =>
        # package installed in the user depot:
        "/home/me/.julia/packages/Pub/oKpw/src/Pub.jl",
    # Zebra:
    (UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"), :Zebra) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Zebra/me9k/src/Zebra.jl",
)
```

This example map includes three different kinds of package locations (the first and third are part of the default load path):

1. The private `Priv` package is "vendored" inside the `App` repository.
2. The public `Priv` and `Zebra` packages are in the system depot, where packages installed and managed by the system administrator live. These are available to all users on the system.
3. The `Pub` package is in the user depot, where packages installed by the user live. These are only available to the user who installed them.

Package directories

Package directories provide a simpler kind of environment without the ability to handle name collisions. In a package directory, the set of top-level packages is the set of subdirectories that "look like" packages. A package `X` is exists in a package directory if the directory contains one of the following "entry point" files:

- `X.jl`
- `X/src/X.jl`
- `X.jl/src/X.jl`

Which dependencies a package in a package directory can import depends on whether the package contains a project file:

- If it has a project file, it can only import those packages which are identified in the `[deps]` section of the project file.
- If it does not have a project file, it can import any top-level package—i.e. the same packages that can be loaded in `Main` or the REPL.

The roots map is determined by examining the contents of the package directory to generate a list of all packages that exist. Additionally, a UUID will be assigned to each entry as follows: For a given package found inside the folder `X`...

1. If `X/Project.toml` exists and has a `uuid` entry, then `uuid` is that value.
2. If `X/Project.toml` exists and but does not have a top-level UUID entry, `uuid` is a dummy UUID generated by hashing the canonical (real) path to `X/Project.toml`.
3. Otherwise (if `Project.toml` does not exist), then `uuid` is the all-zero nil UUID.

The dependency graph of a project directory is determined by the presence and contents of project files in the sub-directory of each package. The rules are:

- If a package subdirectory has no project file, then it is omitted from graph and import statements in its code are treated as top-level, the same as the main project and REPL.
- If a package subdirectory has a project file, then the graph entry for its UUID is the `[deps]` map of the project file, which is considered to be empty if the section is absent.

As an example, suppose a package directory has the following structure and content:

```
Aardvark/
    src/Aardvark.jl:
        import Bobcat
        import Cobra

Bobcat/
    Project.toml:
        [deps]
        Cobra = "4725e24d-f727-424b-bca0-c4307a3456fa"
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"
```

```
    src/Bobcat.jl:
        import Cobra
        import Dingo

Cobra/
    Project.toml:
        uuid = "4725e24d-f727-424b-bca0-c4307a3456fa"
        [deps]
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Cobra.jl:
        import Dingo

Dingo/
    Project.toml:
        uuid = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Dingo.jl:
        # no imports
```

Here is a corresponding roots structure, represented as a dictionary:

```
roots = Dict(
    :Aardvark => UUID("00000000-0000-0000-0000-000000000000"), # no project file, nil UUID
    :Bobcat   => UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), # dummy UUID based on path
    :Cobra    => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), # UUID from project file
    :Dingo    => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), # UUID from project file
)
```

Here is the corresponding graph structure, represented as a dictionary:

```
graph = Dict(
    # Bobcat:
    UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf") => Dict(
        :Cobra => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"),
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Cobra:
```

```
    UUID("4725e24d-f727-424b-bca0-c4307a3456fa") => Dict(
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Dingo:
    UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc") => Dict(),
)
```

A few general rules to note:

1. A package without a project file can depend on any top-level dependency, and since every package in a package directory is available at the top-level, it can import all packages in the environment.
2. A package with a project file cannot depend on one without a project file since packages with project files can only load packages in `graph` and packages without project files do not appear in `graph`.
3. A package with a project file but no explicit UUID can only be depended on by packages without project files since dummy UUIDs assigned to these packages are strictly internal.

Observe the following specific instances of these rules in our example:

- `Aardvark` can import on any of `Bobcat`, `Cobra` or `Dingo`; it does import `Bobcat` and `Cobra`.
- `Bobcat` can and does import both `Cobra` and `Dingo`, which both have project files with UUIDs and are declared as dependencies in `Bobcat`'s `[deps]` section.
- `Bobcat` cannot depend on `Aardvark` since `Aardvark` does not have a project file.
- `Cobra` can and does import `Dingo`, which has a project file and UUID, and is declared as a dependency in `Cobra`'s `[deps]` section.
- `Cobra` cannot depend on `Aardvark` or `Bobcat` since neither have real UUIDs.
- `Dingo` cannot import anything because it has a project file without a `[deps]` section.

The paths map in a package directory is simple: it maps subdirectory names to their corresponding entry-point paths. In other words, if the path to our example project directory is `/home/me/animals` then the `paths` map could be represented by this dictionary:

```
paths = Dict(
    (UUID("00000000-0000-0000-0000-000000000000"), :Aardvark) =>
        "/home/me/AnimalPackages/Aardvark/src/Aardvark.jl",
```

```
    (UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), :Bobcat) =>
        "/home/me/AnimalPackages/Bobcat/src/Bobcat.jl",
    (UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), :Cobra) =>
        "/home/me/AnimalPackages/Cobra/src/Cobra.jl",
    (UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), :Dingo) =>
        "/home/me/AnimalPackages/Dingo/src/Dingo.jl",
)
```

Since all packages in a package directory environment are, by definition, subdirectories with the expected entry-point files, their `paths` map entries always have this form.

Environment stacks

The third and final kind of environment is one that combines other environments by overlaying several of them, making the packages in each available in a single composite environment. These composite environments are called environment stacks. The Julia `LOAD_PATH` global defines an environment stack—the environment in which the Julia process operates. If you want your Julia process to have access only to the packages in one project or package directory, make it the only entry in `LOAD_PATH`. It is often quite useful, however, to have access to some of your favorite tools—standard libraries, profilers, debuggers, personal utilities, etc.—even if they are not dependencies of the project you're working on. By adding an environment containing these tools to the load path, you immediately have access to them in top-level code without needing to add them to your project.

The mechanism for combining the roots, graph and paths data structures of the components of an environment stack is simple: they are merged as dictionaries, favoring earlier entries over later ones in the case of key collisions. In other words, if we have `stack = [env₁, env₂, …]` then we have:

```
roots = reduce(merge, reverse([roots₁, roots₂, …]))
graph = reduce(merge, reverse([graph₁, graph₂, …]))
paths = reduce(merge, reverse([paths₁, paths₂, …]))
```

The subscripted `roots`$_i$, `graph`$_i$ and `paths`$_i$ variables correspond to the subscripted environments, `env`$_i$, contained in `stack`. The `reverse` is present because `merge` favors the last argument rather than first when there are collisions between keys in its argument dictionaries. There are a couple of noteworthy features of this design:

1. The primary environment—i.e. the first environment in a stack—is faithfully embedded in a stacked environment. The full dependency graph of the first environment in a stack is guaranteed to be included intact in the stacked environment including the same versions of all dependencies.

2. Packages in non-primary environments can end up using incompatible versions of their dependencies even if their own environments are entirely compatible. This can happen when one of their dependencies is shadowed by a version in an earlier environment in the stack (either by graph or path, or both).

Since the primary environment is typically the environment of a project you're working on, while environments later in the stack contain additional tools, this is the right trade-off: it's better to break your development tools but keep the project working. When such incompatibilities occur, you'll typically want to upgrade your dev tools to versions that are compatible with the main project.

34.4 Conclusion

Federated package management and precise software reproducibility are difficult but worthy goals in a package system. In combination, these goals lead to a more complex package loading mechanism than most dynamic languages have, but it also yields scalability and reproducibility that is more commonly associated with static languages. Typically, Julia users should be able to use the built-in package manager to manage their projects without needing a precise understanding of these interactions. A call to `Pkg.add("X")` will add to the appropriate project and manifest files, selected via `Pkg.activate("Y")`, so that a future call to `import X` will load `X` without further thought.

Chapter 35

Profiling

The `Profile` module provides tools to help developers improve the performance of their code. When used, it takes measurements on running code, and produces output that helps you understand how much time is spent on individual line(s). The most common usage is to identify "bottlenecks" as targets for optimization.

`Profile` implements what is known as a "sampling" or statistical profiler. It works by periodically taking a backtrace during the execution of any task. Each backtrace captures the currently-running function and line number, plus the complete chain of function calls that led to this line, and hence is a "snapshot" of the current state of execution.

If much of your run time is spent executing a particular line of code, this line will show up frequently in the set of all backtraces. In other words, the "cost" of a given line—or really, the cost of the sequence of function calls up to and including this line—is proportional to how often it appears in the set of all backtraces.

A sampling profiler does not provide complete line-by-line coverage, because the backtraces occur at intervals (by default, 1 ms on Unix systems and 10 ms on Windows, although the actual scheduling is subject to operating system load). Moreover, as discussed further below, because samples are collected at a sparse subset of all execution points, the data collected by a sampling profiler is subject to statistical noise.

Despite these limitations, sampling profilers have substantial strengths:

- You do not have to make any modifications to your code to take timing measurements.
- It can profile into Julia's core code and even (optionally) into C and Fortran libraries.
- By running "infrequently" there is very little performance overhead; while profiling, your code can run at nearly native speed.

For these reasons, it's recommended that you try using the built-in sampling profiler before considering any alternatives.

35.1 Basic usage

Let's work with a simple test case:

```
julia> function myfunc()
           A = rand(200, 200, 400)
           maximum(A)
       end
```

It's a good idea to first run the code you intend to profile at least once (unless you want to profile Julia's JIT-compiler):

```
julia> myfunc() # run once to force compilation
```

Now we're ready to profile this function:

```
julia> using Profile

julia> @profile myfunc()
```

To see the profiling results, there is a graphical browser available, but here we'll use the text-based display that comes with the standard library:

```
julia> Profile.print()
80 ./event.jl:73; (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
 80 ./REPL.jl:97; macro expansion
  80 ./REPL.jl:66; eval_user_input(::Any, ::Base.REPL.REPLBackend)
   80 ./boot.jl:235; eval(::Module, ::Any)
    80 ./<missing>:?; anonymous
     80 ./profile.jl:23; macro expansion
      52 ./REPL[1]:2; myfunc()
       38 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type{B...
        38 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr{F...
       14 ./random.jl:278; rand
        14 ./random.jl:277; rand
         14 ./random.jl:366; rand
          14 ./random.jl:369; rand
      28 ./REPL[1]:3; myfunc()
       28 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinear,...
        3  ./reduce.jl:426; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
        25 ./reduce.jl:428; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
```

Each line of this display represents a particular spot (line number) in the code. Indentation is used to indicate the nested sequence of function calls, with more-indented lines being deeper in the sequence of calls. In each line, the first "field" is the number of backtraces (samples) taken at this line or in any functions executed by this line. The second field is the file name and line number and the third field is the function name. Note that the specific line numbers may change as Julia's code changes; if you want to follow along, it's best to run this example yourself.

In this example, we can see that the top level function called is in the file `event.jl`. This is the function that runs the REPL when you launch Julia. If you examine line 97 of `REPL.jl`, you'll see this is where the function `eval_user_input()` is called. This is the function that evaluates what you type at the REPL, and since we're working interactively these functions were invoked when we entered `@profile myfunc()`. The next line reflects actions taken in the `@profile` macro.

The first line shows that 80 backtraces were taken at line 73 of `event.jl`, but it's not that this line was "expensive" on its own: the third line reveals that all 80 of these backtraces were actually triggered inside its call to `eval_user_input`, and so on. To find out which operations are actually taking the time, we need to look deeper in the call chain.

The first "important" line in this output is this one:

```
52 ./REPL[1]:2; myfunc()
```

`REPL` refers to the fact that we defined `myfunc` in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. The `[1]` shows that the function `myfunc` was the first expression evaluated in this REPL session. Line 2 of `myfunc()` contains the call to `rand`, and there were 52 (out of 80) backtraces that occurred at this line. Below that, you can see a call to `dsfmt_fill_array_close_open!` inside `dSFMT.jl`.

A little further down, you see:

```
28 ./REPL[1]:3; myfunc()
```

Line 3 of `myfunc` contains the call to `maximum`, and there were 28 (out of 80) backtraces taken here. Below that, you can see the specific places in `base/reduce.jl` that carry out the time-consuming operations in the `maximum` function for this type of input data.

Overall, we can tentatively conclude that generating the random numbers is approximately twice as expensive as finding the maximum element. We could increase our confidence in this result by collecting more samples:

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
[....]
 3821 ./REPL[1]:2; myfunc()
  3511 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type...
```

```
  3511 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr...
 310  ./random.jl:278; rand
  [....]
2893 ./REPL[1]:3; myfunc()
 2893 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinea...
  [....]
```

In general, if you have `N` samples collected at a line, you can expect an uncertainty on the order of `sqrt(N)` (barring other sources of noise, like how busy the computer is with other tasks). The major exception to this rule is garbage collection, which runs infrequently but tends to be quite expensive. (Since Julia's garbage collector is written in C, such events can be detected using the `C=true` output mode described below, or by using ProfileView.jl.)

This illustrates the default "tree" dump; an alternative is the "flat" dump, which accumulates counts independent of their nesting:

```
julia> Profile.print(format=:flat)
 Count File          Line Function
  6714 ./<missing>     -1 anonymous
  6714 ./REPL.jl       66 eval_user_input(::Any, ::Base.REPL.REPLBackend)
  6714 ./REPL.jl       97 macro expansion
  3821 ./REPL[1]        2 myfunc()
  2893 ./REPL[1]        3 myfunc()
  6714 ./REPL[7]        1 macro expansion
  6714 ./boot.jl      235 eval(::Module, ::Any)
  3511 ./dSFMT.jl      84 dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_s...
  6714 ./event.jl      73 (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
  6714 ./profile.jl    23 macro expansion
  3511 ./random.jl    431 rand!(::MersenneTwister, ::Array{Float64,3}, ::In...
   310 ./random.jl    277 rand
   310 ./random.jl    278 rand
   310 ./random.jl    366 rand
   310 ./random.jl    369 rand
  2893 ./reduce.jl    270 _mapreduce(::Base.#identity, ::Base.#scalarmax, :...
     5 ./reduce.jl    420 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
   253 ./reduce.jl    426 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
  2592 ./reduce.jl    428 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
    43 ./reduce.jl    429 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
```

If your code has recursion, one potentially-confusing point is that a line in a "child" function can accumulate more counts than there are total backtraces. Consider the following function definitions:

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

If you were to profile **dumbsum3**, and a backtrace was taken while it was executing **dumbsum(1)**, the backtrace would look like this:

```
dumbsum3
    dumbsum(3)
        dumbsum(2)
            dumbsum(1)
```

Consequently, this child function gets 3 counts, even though the parent only gets one. The "tree" representation makes this much clearer, and for this reason (among others) is probably the most useful way to view the results.

35.2 Accumulation and clearing

Results from `@profile` accumulate in a buffer; if you run multiple pieces of code under `@profile`, then `Profile.print()` will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with `Profile.clear()`.

35.3 Options for controlling the display of profile results

`Profile.print` has more options than we've described so far. Let's see the full declaration:

```
function print(io::IO = stdout, data = fetch(); kwargs...)
```

Let's first discuss the two positional arguments, and later the keyword arguments:

- **io** – Allows you to save the results to a buffer, e.g. a file, but the default is to print to **stdout** (the console).
- **data** – Contains the data you want to analyze; by default that is obtained from `Profile.fetch()`, which pulls out the backtraces from a pre-allocated buffer. For example, if you want to profile the profiler, you could say:

  ```
  data = copy(Profile.fetch())
  Profile.clear()
  @profile Profile.print(stdout, data) # Prints the previous results
  Profile.print()                      # Prints results from Profile.print()
  ```

The keyword arguments can be any combination of:

- `format` – Introduced above, determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded). Try running the introductory example with `Profile.print(C = true)`. This can be extremely helpful in deciding whether it's Julia code or C code that is causing a bottleneck; setting `C = true` also improves the interpretability of the nesting, at the cost of longer profile dumps.
- `combine` – Some lines of code contain multiple operations; for example, `s += A[i]` contains both an array reference (`A[i]`) and a sum operation. These correspond to different lines in the generated machine code, and hence there may be two or more different addresses captured during backtraces on this line. `combine = true` lumps them together, and is probably what you typically want, but you can generate an output separately for each unique instruction pointer with `combine = false`.
- `maxdepth` – Limits frames at a depth higher than `maxdepth` in the `:tree` format.
- `sortedby` – Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, whereas `:count` sorts in order of number of collected samples.
- `noisefloor` – Limits frames that are below the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.
- `mincount` – Limits frames with less than `mincount` occurrences.

File/function names are sometimes truncated (with `...`), and indentation is truncated with a `+n` at the beginning, where `n` is the number of extra spaces that would have been inserted, had there been room. If you want a complete profile of deeply-nested code, often a good idea is to save to a file using a wide `displaysize` in an `IOContext`:

```
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

35.4 Configuration

`@profile` just accumulates backtraces, and the analysis happens when you call `Profile.print()`. For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

```
Profile.init() # returns the current settings
Profile.init(n = 10^7, delay = 0.01)
```

`n` is the total number of instruction pointers you can store, with a default value of `10^6`. If your typical backtrace is 20 instruction pointers, then you can collect 50000 backtraces, which suggests a statistical uncertainty of less than 1%. This may be good enough for most applications.

Consequently, you are more likely to need to modify `delay`, expressed in seconds, which sets the amount of time that Julia gets between snapshots to perform the requested computations. A very long-running job might not need frequent backtraces. The default setting is `delay = 0.001`. Of course, you can decrease the delay as well as increase it; however, the overhead of profiling grows once the delay becomes similar to the amount of time needed to take a backtrace (~30 microseconds on the author's laptop).

Chapter 36

Memory allocation analysis

One of the most common techniques to improve performance is to reduce memory allocation. The total amount of allocation can be measured with `@time` and `@allocated`, and specific lines triggering allocation can often be inferred from profiling via the cost of garbage collection that these lines incur. However, sometimes it is more efficient to directly measure the amount of memory allocated by each line of code.

To measure allocation line-by-line, start Julia with the `--track-allocation=<setting>` command-line option, for which you can choose `none` (the default, do not measure allocation), `user` (measure memory allocation everywhere except Julia's core code), or `all` (measure memory allocation at each line of Julia code). Allocation gets measured for each line of compiled code. When you quit Julia, the cumulative results are written to text files with `.mem` appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The `Coverage` package contains some elementary analysis tools, for example to sort the lines in order of number of bytes allocated.

In interpreting the results, there are a few important details. Under the `user` setting, the first line of any function directly called from the REPL will exhibit allocation due to events that happen in the REPL code itself. More significantly, JIT-compilation also adds to allocation counts, because much of Julia's compiler is written in Julia (and compilation usually requires memory allocation). The recommended procedure is to force compilation by executing all the commands you want to analyze, then call `Profile.clear_malloc_data()` to reset all allocation counters. Finally, execute the desired commands and quit Julia to trigger the generation of the `.mem` files.

Chapter 37

External Profiling

Currently Julia supports `Intel VTune`, `OProfile` and `perf` as external profiling tools.

Depending on the tool you choose, compile with `USE_INTEL_JITEVENTS`, `USE_OPROFILE_JITEVENTS` and `USE_PERF_JITEVENTS` set to 1 in `Make.user`. Multiple flags are supported.

Before running Julia set the environment variable `ENABLE_JITPROFILING` to 1.

Now you have a multitude of ways to employ those tools! For example with `OProfile` you can try a simple recording :

```
>ENABLE_JITPROFILING=1 sudo operf -Vdebug ./julia test/fastmath.jl
>opreport -l `which ./julia`
```

Or similary with with `perf` :

```
$ ENABLE_JITPROFILING=1 perf record -o /tmp/perf.data --call-graph dwarf ./julia /test/fastmath.jl
$ perf report --call-graph -G
```

There are many more interesting things that you can measure about your program, to get a comprehensive list please read the Linux perf examples page.

Remember that perf saves for each execution a `perf.data` file that, even for small programs, can get quite large. Also the perf LLVM module saves temporarily debug objects in `~/.debug/jit`, remember to clean that folder frequently.

Chapter 38

Stack Traces

The `StackTraces` module provides simple stack traces that are both human readable and easy to use programmatically.

38.1 Viewing a stack trace

The primary function used to obtain a stack trace is `stacktrace`:

```
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

Calling `stacktrace()` returns a vector of `StackTraces.StackFrame` s. For ease of use, the alias `StackTraces.StackTrace` can be used in place of `Vector{StackFrame}`. (Examples with `[...]` indicate that output may vary depending on how the code is run.)

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
```

```
[...]

julia> @noinline child() = stacktrace()
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> grandparent() = parent()
grandparent (generic function with 1 method)

julia> grandparent()
9-element Array{Base.StackTraces.StackFrame,1}:
 child() at REPL[3]:1
 parent() at REPL[4]:1
 grandparent() at REPL[5]:1
[...]
```

Note that when calling `stacktrace()` you'll typically see a frame with `eval at boot.jl`. When calling `stacktrace()` from the REPL you'll also have a few extra frames in the stack from `REPL.jl`, usually looking something like this:

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

38.2 Extracting useful information

Each `StackTraces.StackFrame` contains the function name, file name, line number, lambda info, a flag indicating whether the frame has been inlined, a flag indicating whether it is a C function (by default C functions do not appear in the stack trace), and an integer representation of the pointer returned by `backtrace`:

```
julia> frame = stacktrace()[3]
eval(::Module, ::Expr) at REPL.jl:5

julia> frame.func
:eval

julia> frame.file
Symbol("~/julia/usr/share/julia/stdlib/v0.7/REPL/src/REPL.jl")

julia> frame.line
5

julia> top_frame.linfo
MethodInstance for eval(::Module, ::Expr)

julia> top_frame.inlined
false

julia> top_frame.from_c
false
```

```
julia> top_frame.pointer
0x00007f92d6293171
```

This makes stack trace information available programmatically for logging, error handling, and more.

38.3 Error handling

While having easy access to information about the current state of the callstack can be helpful in many places, the most obvious application is in error handling and debugging.

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace()
       end
```

```
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[2]:4
 top-level scope
 eval at boot.jl:317 [inlined]
[...]
```

You may notice that in the example above the first stack frame points points at line 4, where `stacktrace` is called, rather than line 2, where bad_function is called, and `bad_function`'s frame is missing entirely. This is understandable, given that `stacktrace` is called from the context of the catch. While in this example it's fairly easy to find the actual source of the error, in complex cases tracking down the source of the error becomes nontrivial.

This can be remedied by passing the result of `catch_backtrace` to `stacktrace`. Instead of returning callstack information for the current context, `catch_backtrace` returns stack information for the context of the most recent exception:

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace(catch_backtrace())
       end
example (generic function with 1 method)

julia> example()
8-element Array{Base.StackTraces.StackFrame,1}:
 bad_function() at REPL[1]:1
 example() at REPL[2]:2
[...]
```

Notice that the stack trace now indicates the appropriate line number and the missing frame.

```
julia> @noinline child() = error("Whoops!")
child (generic function with 1 method)

julia> @noinline parent() = child()
```

```
parent (generic function with 1 method)

julia> @noinline function grandparent()
           try
               parent()
           catch err
               println("ERROR: ", err.msg)
               stacktrace(catch_backtrace())
           end
       end
grandparent (generic function with 1 method)

julia> grandparent()
ERROR: Whoops!
10-element Array{Base.StackTraces.StackFrame,1}:
 error at error.jl:33 [inlined]
 child() at REPL[1]:1
 parent() at REPL[2]:1
 grandparent() at REPL[3]:3
[...]
```

38.4 Exception stacks and `catch_stack`

> Julia 1.1
>
> Exception stacks requires at least Julia 1.1.

While handling an exception further exceptions may be thrown. It can be useful to inspect all these exceptions to identify the root cause of a problem. The julia runtime supports this by pushing each exception onto an internal exception stack as it occurs. When the code exits a `catch` normally, any exceptions which were pushed onto the stack in the associated `try` are considered to be successfully handled and are removed from the stack.

The stack of current exceptions can be accessed using the experimental `Base.catch_stack` function. For example,

```
julia> try
           error("(A) The root cause")
       catch
           try
               error("(B) An exception while handling the exception")
           catch
```

```
            for (exc, bt) in Base.catch_stack()
                showerror(stdout, exc, bt)
                println()
            end
        end
    end
(A) The root cause
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:2
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
(B) An exception while handling the exception
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:5
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
```

In this example the root cause exception (A) is first on the stack, with a further exception (B) following it. After exiting both catch blocks normally (i.e., without throwing a further exception) all exceptions are removed from the stack and are no longer accessible.

The exception stack is stored on the `Task` where the exceptions occurred. When a task fails with uncaught exceptions, `catch_stack(task)` may be used to inspect the exception stack for that task.

38.5 Comparison with `backtrace`

A call to `backtrace` returns a vector of `Union{Ptr{Nothing}, Base.InterpreterIP}`, which may then be passed into `stacktrace` for translation:

```
julia> trace = backtrace()
18-element Array{Union{Ptr{Nothing}, Base.InterpreterIP},1}:
 Ptr{Nothing} @0x00007fd8734c6209
 Ptr{Nothing} @0x00007fd87362b342
 Ptr{Nothing} @0x00007fd87362c136
```

```
 Ptr{Nothing} @0x00007fd87362c986
 Ptr{Nothing} @0x00007fd87362d089
 Base.InterpreterIP(CodeInfo(:(begin
      Core.SSAValue(0) = backtrace()
      trace = Core.SSAValue(0)
      return Core.SSAValue(0)
  end)), 0x0000000000000000)
 Ptr{Nothing} @0x00007fd87362e4cf
[...]

julia> stacktrace(trace)
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

Notice that the vector returned by `backtrace` had 18 elements, while the vector returned by `stacktrace` only has 6. This is because, by default, `stacktrace` removes any lower-level C functions from the stack. If you want to include stack frames from C calls, you can do it like this:

```
julia> stacktrace(trace, true)
21-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167
 do_call at interpreter.c:324
 eval_value at interpreter.c:416
 eval_body at interpreter.c:559
 jl_interpret_toplevel_thunk_callback at interpreter.c:798
 top-level scope
 jl_interpret_toplevel_thunk at interpreter.c:807
 jl_toplevel_eval_flex at toplevel.c:856
 jl_toplevel_eval_in at builtins.c:624
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 jl_apply_generic at gf.c:2167
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 jl_apply_generic at gf.c:2167
```

```
macro expansion at REPL.jl:116 [inlined]
(::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
jl_fptr_trampoline at gf.c:1838
jl_apply_generic at gf.c:2167
jl_apply at julia.h:1540 [inlined]
start_task at task.c:268
ip:0xffffffffffffffff
```

Individual pointers returned by `backtrace` can be translated into `StackTraces.StackFrame` s by passing them into `StackTraces.lookup`:

```
julia> pointer = backtrace()[1];

julia> frame = StackTraces.lookup(pointer)
1-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167

julia> println("The top frame is from $(frame[1].func)!")
The top frame is from jl_apply_generic!
```

Chapter 39

Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

39.1 Avoid global variables

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.

Any code that is performance critical or being benchmarked should be inside a function.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

```
const DEFAULT_VAL = 0
```

Uses of non-constant globals can be optimized by annotating their types at the point of use:

```
global x = rand(1000)

function loop_over_global()
    s = 0.0
    for i in x::Vector{Float64}
        s += i
    end
    return s
end
```

Passing arguments to functions is better style. It leads to more reusable code and clarifies what the inputs and outputs are.

> Note
>
> All code in the REPL is evaluated in global scope, so a variable defined and assigned at top level will be a global variable. Variables defined at top level scope inside modules are also global.

In the following REPL session:

```
julia> x = 1.0
```

is equivalent to:

```
julia> global x = 1.0
```

so all the performance issues discussed previously apply.

39.2 Measure performance with `@time` and pay attention to memory allocation

A useful tool for measuring performance is the `@time` macro. We here repeat the example with the global variable above, but this time with the type annotation removed:

```
julia> x = rand(1000);

julia> function sum_global()
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_global()
  0.017705 seconds (15.28 k allocations: 694.484 KiB)
496.84883432553846

julia> @time sum_global()
  0.000140 seconds (3.49 k allocations: 70.313 KiB)
496.84883432553846
```

On the first call (`@time sum_global()`) the function gets compiled. (If you've not yet used `@time` in this session, it will also compile functions needed for timing.) You should not take the results of this run seriously. For the second run, note that in addition to reporting the time, it also indicated that a significant amount of memory was allocated. We are here just computing a sum over all elements in a vector of 64-bit floats so there should be no need to allocate memory (at least not on the heap which is what `@time` reports).

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability or creating many small temporary arrays. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal. Take such indications seriously and follow the advice below.

If we instead pass `x` as an argument to the function it no longer allocates memory (the allocation reported below is due to running the `@time` macro in global scope) and is significantly faster after the first call:

```
julia> x = rand(1000);

julia> function sum_arg(x)
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_arg(x)
  0.007701 seconds (821 allocations: 43.059 KiB)
496.84883432553846

julia> @time sum_arg(x)
  0.000006 seconds (5 allocations: 176 bytes)
496.84883432553846
```

The 5 allocations seen are from running the `@time` macro itself in global scope. If we instead run the timing in a function, we can see that indeed no allocations are performed:

```
julia> time_sum(x) = @time sum_arg(x);

julia> time_sum(x)
  0.000001 seconds
496.84883432553846
```

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the tools below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see Pre-allocating outputs).

> Note
>
> For more serious benchmarking, consider the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

39.3 Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

- Profiling allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the ProfileView package can help you visualize your profiling results.
- The Traceur package can help you find common performance problems in your code.
- Unexpectedly-large memory allocations–as reported by `@time`, `@allocated`, or the profiler (through calls to the garbage-collection routines)–hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the `--track-allocation=user` option and examine the resulting `*.mem` files to see information about where those allocations occur. See Memory allocation analysis.
- `@code_warntype` generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See `@code_warntype` below.

39.4 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```
julia> a = Real[]
0-element Array{Real,1}

julia> push!(a, 1); push!(a, 2.0); push!(a, π)
3-element Array{Real,1}:
```

```
1
2.0
π
```

Because `a` is a an array of abstract type `Real`, it must be able to hold any `Real` value. Since `Real` objects can be of arbitrary size and structure, `a` must be represented as an array of pointers to individually allocated `Real` objects. However, if we instead only allow numbers of the same type, e.g. `Float64`, to be stored in `a` these can be stored more efficiently:

```
julia> a = Float64[]
0-element Array{Float64,1}

julia> push!(a, 1); push!(a, 2.0); push!(a,  π)
3-element Array{Float64,1}:
 1.0
 2.0
 3.141592653589793
```

Assigning numbers into `a` will now convert them to `Float64` and `a` will be stored as a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under Parametric Types.

39.5 Type declarations

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is not the case in Julia. In Julia, the compiler generally knows the types of all function arguments, local variables, and expressions. However, there are a few specific instances where declarations are helpful.

Avoid fields with abstract type

Types can be declared without specifying the types of their fields:

```
julia> struct MyAmbiguousType
           a
       end
```

This allows `a` to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of

objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType

julia> typeof(c)
MyAmbiguousType
```

The values of `b` and `c` have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field `a`, the fact that the memory representation of a `UInt8` differs from a `Float64` also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of `a`. Here, we are focused on the case where `a` might be any one of several types, in which case the natural solution is to use parameters. For example:

```
julia> mutable struct MyType{T<:AbstractFloat}
           a::T
       end
```

This is a better choice than

```
julia> mutable struct MyStillAmbiguousType
           a::AbstractFloat
       end
```

because the first version specifies the type of `a` from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)
```

```
julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64}

julia> typeof(t)
MyStillAmbiguousType
```

The type of field `a` can be readily determined from the type of `m`, but not from the type of `t`. Indeed, in `t` it's possible to change the type of the field `a`:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once `m` is constructed, the type of `m.a` cannot change:

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float64
```

The fact that the type of `m.a` is known from `m`'s type—coupled with the fact that its type cannot change mid-function—allows the compiler to generate highly-optimized code for objects like `m` but not for objects like `t`.

Of course, all of this is true only if we construct `m` with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
julia> m = MyType{AbstractFloat}(3.2)
MyType{AbstractFloat}(3.2)

julia> typeof(m.a)
```

```
Float64

julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float32
```

For all practical purposes, such objects behave identically to those of `MyStillAmbiguousType`.

It's quite instructive to compare the sheer amount code generated for a simple function

```
func(m::MyType) = m.a+1
```

using

```
code_llvm(func, Tuple{MyType{Float64}})
code_llvm(func, Tuple{MyType{AbstractFloat}})
```

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

Avoid fields with abstract containers

The same best practices also work for container types:

```
julia> struct MySimpleContainer{A<:AbstractVector}
           a::A
       end

julia> struct MyAmbiguousContainer{T}
           a::AbstractVector{T}
       end
```

For example:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}}

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Array{Int64,1}}

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);

julia> typeof(b)
MyAmbiguousContainer{Int64}
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where you might wish that your code could do different things depending on the element type of `a`. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:

```
julia> function sumfoo(c::MySimpleContainer)
           s = 0
           for x in c.a
               s += foo(x)
           end
           s
       end
sumfoo (generic function with 1 method)

julia> foo(x::Integer) = x
foo (generic function with 1 method)
```

```
julia> foo(x::AbstractFloat) = round(x)
foo (generic function with 2 methods)
```

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types or types of the `AbstractVector` of the field `a` in `MySimpleContainer`. You could do it like this:

```
julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:Integer}})
           return c.a[1]+1
       end
myfunc (generic function with 1 method)

julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:AbstractFloat}})
           return c.a[1]+2
       end
myfunc (generic function with 2 methods)

julia> function myfunc(c::MySimpleContainer{Vector{T}}) where T <: Integer
           return c.a[1]+3
       end
myfunc (generic function with 3 methods)
```

```
julia> myfunc(MySimpleContainer(1:3))
2

julia> myfunc(MySimpleContainer(1.0:3))
3.0

julia> myfunc(MySimpleContainer([1:3;]))
4
```

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type (arrays of type `Array{Any}`). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
```

```
    b = x+1
    ...
end
```

Here, we happened to know that the first element of `a` would be an `Int32`. Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

In the case that the type of `a[1]` is not known precisely, `x` can be declared via `x = convert(Int32, a[1])::Int32`. The use of the `convert` function allows `a[1]` to be any object convertible to an `Int32` (such as `UInt8`), thus increasing the genericity of the code by loosening the type requirement. Notice that `convert` itself needs a type annotation in this context in order to achieve type stability. This is because the compiler cannot deduce the type of the return value of a function, even `convert`, unless the types of all the function's arguments are known.

Type annotation will not enhance (and can actually hinder) performance if the type is constructed at run-time. This is because the compiler cannot use the annotation to specialize the subsequent code, and the type-check itself takes time. For example, in the code:

```
function nr(a, prec)
    ctype = prec == 32 ? Float32 : Float64
    b = Complex{ctype}(a)
    c = (b + 1.0f0)::Complex{ctype}
    abs(c)
end
```

the annotation of `c` harms performance. To write performant code involving types constructed at run-time, use the function-barrier technique discussed below, and ensure that the constructed type appears among the argument types of the kernel function so that the kernel operations are properly specialized by the compiler. For example, in the above snippet, as soon as `b` is constructed, it can be passed to another function `k`, the kernel. If, for example, function `k` declares `b` as an argument of type `Complex{T}`, where `T` is a type parameter, then a type annotation appearing in an assignment statement within `k` of the form:

```
c = (b + 1.0f0)::Complex{T}
```

does not hinder performance (but does not help either) since the compiler can determine the type of `c` at the time `k` is compiled.

39.6 Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a "compound function" that should really be written as multiple definitions:

```
using LinearAlgebra

function mynorm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svdvals(A))
    else
        error("mynorm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
norm(x::Vector) = sqrt(real(dot(x, x)))
norm(A::Matrix) = maximum(svdvals(A))
```

It should however be noted that the compiler is quite efficient at optimizing away the dead branches in code written as the `mynorm` example.

39.7 Write "type-stable" functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

```
pos(x) = x < 0 ? 0 : x
```

Although this seems innocent enough, the problem is that `0` is an integer (of type `Int`) and `x` might be of any type. Thus, depending on the value of `x`, this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

```
pos(x) = x < 0 ? zero(x) : x
```

There is also a `oneunit` function, and a more general `oftype(x, y)` function, which returns `y` converted to the type of `x`.

39.8 Avoid changing the type of a variable

An analogous "type-stability" problem exists for variables used repeatedly within a function:

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    return x
end
```

Local variable `x` starts as an integer, and after one loop iteration becomes a floating-point number (the result of `/` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize `x` with `x = 1.0`
- Declare the type of `x`: `x::Float64 = 1`
- Use an explicit conversion: `x = oneunit(Float64)`
- Initialize with the first loop iteration, to `x = 1 / rand()`, then loop `for i = 2:10`

39.9 Separate kernel functions (aka, function barriers)

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

```
julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           for i = 1:n
               a[i] = 2
           end
           return a
       end;
```

```
julia> strange_twos(3)
3-element Array{Float64,1}:
 2.0
 2.0
 2.0
```

This should be written as:

```
julia> function fill_twos!(a)
           for i = eachindex(a)
               a[i] = 2
           end
       end;

julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           fill_twos!(a)
           return a
       end;

julia> strange_twos(3)
3-element Array{Float64,1}:
 2.0
 2.0
 2.0
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of `a` during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of `fill_twos!` for different types of `a`.

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in Julia Base. For example, see `vcat` and `hcat` in `abstractarray.jl`, or the `fill!` function, which we could have used instead of writing our own `fill_twos!`.

Functions like `strange_twos` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

39.10 Types with values-as-parameters

Let's say you want to create an `N`-dimensional array that has size 3 along each axis. Such arrays can be created like this:

```
julia> A = fill(5.0, (3, 3))
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This approach works very well: the compiler can figure out that `A` is an `Array{Float64,2}` because it knows the type of the fill value (`5.0::Float64`) and the dimensionality (`(3, 3)::NTuple{2,Int}`). This implies that the compiler can generate very efficient code for any future usage of `A` in the same function.

But now let's say you want to write a function that creates a 3×3×... array in arbitrary dimensions; you might be tempted to write a function

```
julia> function array3(fillval, N)
           fill(fillval, ntuple(d->3, N))
       end
array3 (generic function with 1 method)

julia> array3(5.0, 2)
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This works, but (as you can verify for yourself using `@code_warntype array3(5.0, 2)`) the problem is that the output type cannot be inferred: the argument `N` is a value of type `Int`, and type-inference does not (and cannot) predict its value in advance. This means that code using the output of this function has to be conservative, checking the type on each access of `A`; such code will be very slow.

Now, one very good way to solve such problems is by using the function-barrier technique. However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through `Val{T}()` (see "Value types"):

```
julia> function array3(fillval, ::Val{N}) where N
           fill(fillval, ntuple(d->3, Val(N)))
```

```
    end
array3 (generic function with 1 method)

julia> array3(5.0, Val(2))
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia has a specialized version of `ntuple` that accepts a `Val{::Int}` instance as the second parameter; by passing `N` as a type-parameter, you make its "value" known to the compiler. Consequently, this version of `array3` allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called `array3` from a function like this:

```
function call_array3(fillval, n)
    A = array3(fillval, Val(n))
end
```

Here, you've created the same problem all over again: the compiler can't guess what `n` is, so it doesn't know the type of `Val(n)`. Attempting to use `Val`, but doing so incorrectly, can easily make performance worse in many situations. (Only in situations where you're effectively combining `Val` with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

An example of correct usage of `Val` would be:

```
function filter3(A::AbstractArray{T,N}) where {T,N}
    kernel = array3(1, Val(N))
    filter(A, kernel)
end
```

In this example, `N` is passed as a parameter, so its "value" is known to the compiler. Essentially, `Val(T)` works only when `T` is either hard-coded/literal (`Val(3)`) or already specified in the type-domain.

39.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there's an understandable tendency to go overboard and try to use it for everything. For example, you might imagine using it to store information, e.g.

```
struct Car{Make, Model}
    year::Int
    ...more fields...
end
```

and then dispatch on objects like `Car{:Honda,:Accord}(year, args...)`.

This might be worthwhile when either of the following are true:

- You require CPU-intensive processing on each `Car`, and it becomes vastly more efficient if you know the `Make` and `Model` at compile time and the total number of different `Make` or `Model` that will be used is not too large.
- You have homogenous lists of the same type of `Car` to process, so that you can store them all in an `Array{Car{:Honda,:Accord},N}`.

When the latter holds, a function processing such a homogenous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can "look up" the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it's likely that you'll get no benefit; worse, the resulting "combinatorial explosion of types" will be counterproductive. If `items[i+1]` has a different type than `item[i]`, Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you're asking the full type- system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found on the mailing list.

Perhaps even worse than the run-time impact is the compile-time impact: Julia will compile specialized functions for each different `Car{Make, Model}`; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom `get_year` function you might write yourself, to the generic `push!` function in Julia Base) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

39.12 Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the `vec` function or the syntax `[:]` as shown below (notice that the array is ordered `[1 3 2 4]`, not `[1 2 3 4]`):

```
julia> x = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> x[:]
4-element Array{Int64,1}:
 1
 3
 2
 4
```

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python (`numpy`) among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the inner-most loop index is the first to appear in a slice expression.

Consider the following contrived example. Imagine we wanted to write a function that accepts a `Vector` and returns a square `Matrix` with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in `repeat`):

```
function copy_cols(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[:, i] = x
    end
    return out
end

function copy_rows(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[i, :] = x
```

```
    end
    return out
end

function copy_col_row(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for col = inds, row = inds
        out[row, col] = x[row]
    end
    return out
end

function copy_row_col(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for row = inds, col = inds
        out[row, col] = x[col]
    end
    return out
end
```

Now we will time each of these functions using the same random `10000` by `1` input vector:

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, [copy_cols, copy_rows, copy_col_row, copy_row_col]);
copy_cols:    0.331706323
copy_rows:    1.799009911
copy_col_row: 0.415630047
copy_row_col: 1.721531501
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

39.13 Pre-allocating outputs

If your function returns an `Array` or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by preallocating the output. As a trivial example, compare

```
julia> function xinc(x)
           return [x, x+1, x+2]
       end;

julia> function loopinc()
           y = 0
           for i = 1:10^7
               ret = xinc(i)
               y += ret[2]
           end
           return y
       end;
```

with

```
julia> function xinc!(ret::AbstractVector{T}, x::T) where T
           ret[1] = x
           ret[2] = x+1
           ret[3] = x+2
           nothing
       end;

julia> function loopinc_prealloc()
           ret = Vector{Int}(undef, 3)
           y = 0
           for i = 1:10^7
               xinc!(ret, i)
               y += ret[2]
           end
           return y
       end;
```

Timing results:

```
julia> @time loopinc()
  0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc time)
50000015000000

julia> @time loopinc_prealloc()
  0.030850 seconds (6 allocations: 288 bytes)
50000015000000
```

Preallocation has other advantages, for example by allowing the caller to control the "output" type from an algorithm. In the example above, we could have passed a `SubArray` rather than an `Array`, had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required. However, for "vectorized" (element-wise) functions, the convenient syntax `x .= f.(y)` can be used for in-place operations with fused loops and no temporary arrays (see the dot syntax for vectorizing functions).

39.14 More dots: Fuse vectorized operations

Julia has a special dot syntax that converts any scalar function into a "vectorized" function call, and any operator into a "vectorized" operator, with the special property that nested "dot calls" are fusing: they are combined at the syntax level into a single loop, without allocating temporary arrays. If you use `.=` and similar assignment operators, the result can also be stored in-place in a pre-allocated array (see above).

In a linear-algebra context, this means that even though operations like `vector + vector` and `vector * scalar` are defined, it can be advantageous to instead use `vector .+ vector` and `vector .* scalar` because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

```
julia> f(x) = 3x.^2 + 4x + 7x.^3;

julia> fdot(x) = @. 3x^2 + 4x + 7x^3 # equivalent to 3 .* x.^2 .+ 4 .* x .+ 7 .* x.^3;
```

Both `f` and `fdot` compute the same thing. However, `fdot` (defined with the help of the `@.` macro) is significantly faster when applied to an array:

```
julia> x = rand(10^6);

julia> @time f(x);
  0.019049 seconds (16 allocations: 45.777 MiB, 18.59% gc time)
```

```
julia> @time fdot(x);
  0.002790 seconds (6 allocations: 7.630 MiB)

julia> @time f.(x);
  0.002626 seconds (8 allocations: 7.630 MiB)
```

That is, `fdot(x)` is ten times faster and allocates 1/6 the memory of `f(x)`, because each `*` and `+` operation in `f(x)` allocates a new temporary array and executes in a separate loop. (Of course, if you just do `f.(x)` then it is as fast as `fdot(x)` in this example, but in many contexts it is more convenient to just sprinkle some dots in your expressions rather than defining a separate function for each vectorized operation.)

39.15 Consider using views for slices

In Julia, an array "slice" expression like `array[1:5, :]` creates a copy of that data (except on the left-hand side of an assignment, where `array[1:5, :] = ...` assigns in-place to that portion of `array`). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a `SubArray`) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling `view`, or more simply for a whole expression or block of code by putting `@views` in front of that expression. For example:

```
julia> fcopy(x) = sum(x[2:end-1]);

julia> @views fview(x) = sum(x[2:end-1]);

julia> x = rand(10^6);

julia> @time fcopy(x);
  0.003051 seconds (7 allocations: 7.630 MB)

julia> @time fview(x);
  0.001020 seconds (6 allocations: 224 bytes)
```

Notice both the 3× speedup and the decreased memory allocation of the `fview` version of the function.

39.16 Copying data is not always bad

Arrays are stored contiguously in memory, lending themselves to CPU vectorization and fewer memory accesses due to caching. These are the same reasons that it is recommended to access arrays in column-major order (see above). Irregular access patterns and non-contiguous views can drastically slow down computations on arrays because of non-sequential memory access.

Copying irregularly-accessed data into a contiguous array before operating on it can result in a large speedup, such as in the example below. Here, a matrix and a vector are being accessed at 800,000 of their randomly-shuffled indices before being multiplied. Copying the views into plain arrays speeds up the multiplication even with the cost of the copying operation.

```
julia> using Random

julia> x = randn(1_000_000);

julia> inds = shuffle(1:1_000_000)[1:800000];

julia> A = randn(50, 1_000_000);

julia> xtmp = zeros(800_000);

julia> Atmp = zeros(50, 800_000);

julia> @time sum(view(A, :, inds) * view(x, inds))
  0.412156 seconds (14 allocations: 960 bytes)
-4256.759568345458

julia> @time begin
           copyto!(xtmp, view(x, inds))
           copyto!(Atmp, view(A, :, inds))
           sum(Atmp * xtmp)
       end
  0.285923 seconds (14 allocations: 960 bytes)
-4256.759568345134
```

Provided there is enough memory for the copies, the cost of copying the view to an array is far outweighed by the speed boost from doing the matrix multiplication on a contiguous array.

39.17 Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
println(file, "$a $b")
```

use:

```
println(file, a, " ", b)
```

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

```
println(file, "$(f(a))$(f(b))")
```

versus:

```
println(file, f(a), f(b))
```

39.18 Optimize network I/O during parallel execution

When executing a remote function in parallel:

```
using Distributed

responses = Vector{Any}(undef, nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(foo, pid, args...)
    end
end
```

is faster than:

```
using Distributed

refs = Vector{Any}(undef, nworkers())
```

```
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

The former results in a single network round-trip to every worker, while the latter results in two network calls - first by the `@spawnat` and the second due to the `fetch` (or even a `wait`). The `fetch`/`wait` is also being executed serially resulting in an overall poorer performance.

39.19 Fix deprecation warnings

A deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings.

39.20 Tweaks

These are some minor points that might help in tight inner loops.

- Avoid unnecessary arrays. For example, instead of `sum([x,y,z])` use `x+y+z`.
- Use `abs2(z)` instead of `abs(z)^2` for complex `z`. In general, try to rewrite code to use `abs2` instead of `abs` for complex arguments.
- Use `div(x,y)` for truncating division of integers instead of `trunc(x/y)`, `fld(x,y)` instead of `floor(x/y)`, and `cld(x,y)` instead of `ceil(x/y)`.

39.21 Performance Annotations

Sometimes you can enable better optimization by promising certain program properties.

- Use `@inbounds` to eliminate array bounds checking within expressions. Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption.
- Use `@fastmath` to allow floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers. Be careful when doing this, as this may change numerical results. This corresponds to the `-ffast-math` option of clang.
- Write `@simd` in front of `for` loops to promise that the iterations are independent and may be reordered. Note that in many cases, Julia can automatically vectorize code without the `@simd` macro; it is only beneficial in

cases where such a transformation would otherwise be illegal, including cases like allowing floating-point re-associativity and ignoring dependent memory accesses (`@simd ivdep`). Again, be very careful when asserting `@simd` as erroneously annotating a loop with dependent iterations may result in unexpected results. In particular, note that `setindex!` on some `AbstractArray` subtypes is inherently dependent upon iteration order. This feature is experimental and could change or disappear in future versions of Julia.

The common idiom of using 1:n to index into an AbstractArray is not safe if the Array uses unconventional indexing, and may cause a segmentation fault if bounds checking is turned off. Use `LinearIndices(x)` or `eachindex(x)` instead (see also offset-arrays).

Note

While `@simd` needs to be placed directly in front of an innermost `for` loop, both `@inbounds` and `@fastmath` can be applied to either single expressions or all the expressions that appear within nested blocks of code, e.g., using `@inbounds begin` or `@inbounds for ...`.

Here is an example with both `@inbounds` and `@simd` markup (we here use `@noinline` to prevent the optimizer from trying to be too clever and defeat our benchmark):

```
@noinline function inner(x, y)
    s = zero(eltype(x))
    for i=eachindex(x)
        @inbounds s += x[i]*y[i]
    end
    return s
end

@noinline function innersimd(x, y)
    s = zero(eltype(x))
    @simd for i = eachindex(x)
        @inbounds s += x[i] * y[i]
    end
    return s
end

function timeit(n, reps)
    x = rand(Float32, n)
    y = rand(Float32, n)
    s = zero(Float64)
```

```
    time = @elapsed for j in 1:reps
        s += inner(x, y)
    end
    println("GFlop/sec        = ", 2n*reps / time*1E-9)
    time = @elapsed for j in 1:reps
        s += innersimd(x, y)
    end
    println("GFlop/sec (SIMD) = ", 2n*reps / time*1E-9)
end

timeit(1000, 1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
GFlop/sec        = 1.9467069505224963
GFlop/sec (SIMD) = 17.578554163920018
```

(`GFlop/sec` measures the performance, and larger numbers are better.)

Here is an example with all three kinds of markup. This program first calculates the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

```
function init!(u::Vector)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds @simd for i in 1:n #by asserting that `u` is a `Vector` we can assume it has 1-based
    ↪  indexing
        u[i] = sin(2pi*dx*i)
    end
end

function deriv!(u::Vector, du)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds du[1] = (u[2] - u[1]) / dx
    @fastmath @inbounds @simd for i in 2:n-1
        du[i] = (u[i+1] - u[i-1]) / (2*dx)
    end
    @fastmath @inbounds du[n] = (u[n] - u[n-1]) / dx
end
```

```
function mynorm(u::Vector)
    n = length(u)
    T = eltype(u)
    s = zero(T)
    @fastmath @inbounds @simd for i in 1:n
        s += u[i]^2
    end
    @fastmath @inbounds return sqrt(s)
end

function main()
    n = 2000
    u = Vector{Float64}(undef, n)
    init!(u)
    du = similar(u)

    deriv!(u, du)
    nu = mynorm(du)

    @time for i in 1:10^6
        deriv!(u, du)
        nu = mynorm(du)
    end

    println(nu)
end

main()
```

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

```
$ julia wave.jl;
  1.207814709 seconds
4.443986180758249

$ julia --math-mode=ieee wave.jl;
  4.487083643 seconds
4.443986180758249
```

Here, the option `--math-mode=ieee` disables the `@fastmath` macro, so that we can compare results.

In this case, the speedup due to `@fastmath` is a factor of about 3.7. This is unusually large – in general, the speedup will be smaller. (In this particular example, the working set of the benchmark is small enough to fit into the L1 cache of the processor, so that memory access latency does not play a role, and computing time is dominated by CPU usage. In many real world programs this is not the case.) Also, in this case this optimization does not change the result – in general, the result will be slightly different. In some cases, especially for numerically unstable algorithms, the result can be very different.

The annotation `@fastmath` re-arranges floating point expressions, e.g. changing the order of evaluation, or assuming that certain special cases (inf, nan) cannot occur. In this case (and on this particular computer), the main difference is that the expression `1 / (2*dx)` in the function `deriv` is hoisted out of the loop (i.e. calculated outside the loop), as if one had written `idx = 1 / (2*dx)`. In the loop, the expression `... / (2*dx)` then becomes `... * idx`, which is much faster to evaluate. Of course, both the actual optimization that is applied by the compiler as well as the resulting speedup depend very much on the hardware. You can examine the change in generated code by using Julia's `code_native` function.

Note that `@fastmath` also assumes that `NaN`s will not occur during the computation, which can lead to surprising behavior:

```
julia> f(x) = isnan(x);

julia> f(NaN)
true

julia> f_fast(x) = @fastmath isnan(x);

julia> f_fast(NaN)
false
```

39.22 Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called denormal numbers, are useful in many contexts, but incur a performance penalty on some hardware. A call `set_zero_subnormals(true)` grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call `set_zero_subnormals(false)` enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

```
function timestep(b::Vector{T}, a::Vector{T}, Δt::T) where T
    @assert length(a)==length(b)
```

```
    n = length(b)
    b[1] = 1                           # Boundary condition
    for i=2:n-1
        b[i] = a[i] + (a[i-1] - T(2)*a[i] + a[i+1]) * Δt
    end
    b[n] = 0                           # Boundary condition
end

function heatflow(a::Vector{T}, nstep::Integer) where T
    b = similar(a)
    for t=1:div(nstep,2)               # Assume nstep is even
        timestep(b,a,T(0.1))
        timestep(a,b,T(0.1))
    end
end

heatflow(zeros(Float32,10),2)          # Force compilation
for trial=1:6
    a = zeros(Float32,1000)
    set_zero_subnormals(iseven(trial))  # Odd trials use strict IEEE arithmetic
    @time heatflow(a,1000)
end
```

This gives an output similar to

```
  0.002202 seconds (1 allocation: 4.063 KiB)
  0.001502 seconds (1 allocation: 4.063 KiB)
  0.002139 seconds (1 allocation: 4.063 KiB)
  0.001454 seconds (1 allocation: 4.063 KiB)
  0.002115 seconds (1 allocation: 4.063 KiB)
  0.001455 seconds (1 allocation: 4.063 KiB)
```

Note how each even iteration is significantly faster.

This example generates many subnormal numbers because the values in `a` become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as `x-y == 0` implies `x == y`:

```
julia> x = 3f-38; y = 2f-38;
```

```
julia> set_zero_subnormals(true); (x - y, x == y)
(0.0f0, false)

julia> set_zero_subnormals(false); (x - y, x == y)
(1.0000001f-38, false)
```

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing `a` with zeros, initialize it with:

```
a = rand(Float32,1000) * 1.f-9
```

39.23 @code_warntype

The macro `@code_warntype` (or its function variant `code_warntype`) can sometimes be helpful in diagnosing type-related problems. Here's an example:

```
julia> @noinline pos(x) = x < 0 ? 0 : x;

julia> function f(x)
           y = pos(x)
           return sin(y*x + 1)
       end;

julia> @code_warntype f(3.2)
Variables
  #self#::Core.Compiler.Const(f, false)
  x::Float64
  y::Union{Float64, Int64}

Body::Float64
1 —      (y = Main.pos(x))
│   %2 = (y * x)::Float64
│   %3 = (%2 + 1)::Float64
│   %4 = Main.sin(%3)::Float64
└──      return %4
```

Interpreting the output of `@code_warntype`, like that of its cousins `@code_lowered`, `@code_typed`, `@code_llvm`, and `@code_native`, takes a little practice. Your code is being presented in form that has been heavily digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the `::T`

(where `T` might be `Float64`, for example). The most important characteristic of `@code_warntype` is that non-concrete types are displayed in red; in the above example, such output is shown in uppercase.

At the top, the inferred return type of the function is shown as `Body::Float64`. The next lines represent the body of `f` in Julia's SSA IR form. The numbered boxes are labels and represent targets for jumps (via `goto`) in your code. Looking at the body, you can see that the first thing that happens is that `pos` is called and the return value has been inferred as the `Union` type `UNION{FLOAT64, INT64}` shown in uppercase since it is a non-concrete type. This means that we cannot know the exact return type of `pos` based on the input types. However, the result of `y*x`is a `Float64` no matter if `y` is a `Float64` or `Int64` The net result is that `f(x::Float64)` will not be type-unstable in its output, even if some of the intermediate computations are type-unstable.

How you use this information is up to you. Obviously, it would be far and away best to fix `pos` to be type-stable: if you did so, all of the variables in `f` would be concrete, and its performance would be optimal. However, there are circumstances where this kind of ephemeral type instability might not matter too much: for example, if `pos` is never used in isolation, the fact that `f`'s output is type-stable (for `Float64` inputs) will shield later code from the propagating effects of type instability. This is particularly relevant in cases where fixing the type instability is difficult or impossible. In such cases, the tips above (e.g., adding type annotations and/or breaking up functions) are your best tools to contain the "damage" from type instability. Also, note that even Julia Base has functions that are type unstable. For example, the function `findfirst` returns the index into an array where a key is found, or `nothing` if it is not found, a clear type instability. In order to make it easier to find the type instabilities that are likely to be important, `Union`s containing either `missing` or `nothing` are color highlighted in yellow, instead of red.

The following examples may help you interpret expressions marked as containing non-leaf types:

- Function body starting with `Body::UNION{T1,T2})`
  - Interpretation: function with unstable return type
  - Suggestion: make the return value type-stable, even if you have to annotate it
- `invoke Main.g(%%x::Int64)::UNION{FLOAT64, INT64}`
  - Interpretation: call to a type-unstable function `g`.
  - Suggestion: fix the function, or if necessary annotate the return value
- `invoke Base.getindex(%%x::Array{Any,1}, 1::Int64)::ANY`
  - Interpretation: accessing elements of poorly-typed arrays
  - Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses

- `Base.getfield(%%x, :(:data))::ARRAY{FLOAT64,N} WHERE N`
  - Interpretation: getting a field that is of non-leaf type. In this case, `ArrayContainer` had a field `data::Array{T}`. But `Array` needs the dimension `N`, too, to be a concrete type.
  - Suggestion: use concrete types like `Array{T,3}` or `Array{T,N}`, where `N` is now a parameter of `ArrayContainer`

39.24 Performance of captured variable

Consider the following example that defines an inner function:

```
function abmult(r::Int)
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

Function `abmult` returns a function `f` that multiplies its argument by the absolute value of `r`. The inner function assigned to `f` is called a "closure". Inner functions are also used by the language for `do`-blocks and for generator expressions.

This style of code presents performance challenges for the language. The parser, when translating it into lower-level instructions, substantially reorganizes the above code by extracting the inner function to a separate code block. "Captured" variables such as `r` that are shared by inner functions and their enclosing scope are also extracted into a heap-allocated "box" accessible to both inner and outer functions because the language specifies that `r` in the inner scope must be identical to `r` in the outer scope even after the outer scope (or another inner function) modifies `r`.

The discussion in the preceding paragraph referred to the "parser", that is, the phase of compilation that takes place when the module containing `abmult` is first loaded, as opposed to the later phase when it is first invoked. The parser does not "know" that `Int` is a fixed type, or that the statement `r = -r` transforms an `Int` to another `Int`. The magic of type inference takes place in the later phase of compilation.

Thus, the parser does not know that `r` has a fixed type (`Int`). nor that `r` does not change value once the inner function is created (so that the box is unneeded). Therefore, the parser emits code for box that holds an object with an abstract type such as `Any`, which requires run-time type dispatch for each occurrence of `r`. This can be verified by applying `@code_warntype` to the above function. Both the boxing and the run-time type dispatch can cause loss of performance.

If captured variables are used in a performance-critical section of the code, then the following tips help ensure that their use is performant. First, if it is known that a captured variable does not change its type, then this can be declared explicitly with a type annotation (on the variable, not the right-hand side):

```
function abmult2(r0::Int)
    r::Int = r0
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

The type annotation partially recovers lost performance due to capturing because the parser can associate a concrete type to the object in the box. Going further, if the captured variable does not need to be boxed at all (because it will not be reassigned after the closure is created), this can be indicated with `let` blocks as follows.

```
function abmult3(r::Int)
    if r < 0
        r = -r
    end
    f = let r = r
            x -> x * r
    end
    return f
end
```

The `let` block creates a new variable `r` whose scope is only the inner function. The second technique recovers full language performance in the presence of captured variables. Note that this is a rapidly evolving aspect of the compiler, and it is likely that future releases will not require this degree of programmer annotation to attain performance. In the mean time, some user-contributed packages like FastClosures automate the insertion of `let` statements as in `abmult3`.

Chapter 40

Workflow Tips

Here are some tips for working with Julia efficiently.

40.1 REPL-based workflow

As already elaborated in The Julia REPL, Julia's REPL provides rich functionality that facilitates an efficient interactive workflow. Here are some tips that might further enhance your experience at the command line.

A basic editor/REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the `julia` command line. A common pattern includes the following elements:

- Put code under development in a temporary module. Create a file, say `Tmp.jl`, and include within it

```
module Tmp
export say_hello

say_hello() = println("Hello!")

# your other definitions here

end
```

- Put your test code in another file. Create another file, say `tst.jl`, which looks like

```
include("Tmp.jl")
import .Tmp
# using .Tmp # we can use `using` to bring the exported symbols in `Tmp` into our namespace
```

```
Tmp.say_hello()
# say_hello()

# your other test code here
```

and includes tests for the contents of `Tmp`. Alternatively, you can wrap the contents of your test file in a module, as

```
module Tst
    include("Tmp.jl")
    import .Tmp
    #using .Tmp

    Tmp.say_hello()
    # say_hello()

    # your other test code here
end
```

The advantage is that your testing code is now contained in a module and does not use the global scope in `Main` for definitions, which is a bit more tidy.

- `include` the `tst.jl` file in the Julia REPL with `include("tst.jl")`.
- Lather. Rinse. Repeat. Explore ideas at the `julia` command prompt. Save good ideas in `tst.jl`. To execute `tst.jl` after it has been changed, just `include` it again.

40.2 Browser-based workflow

It is also possible to interact with a Julia REPL in the browser via IJulia. See the package home for details.

40.3 Revise-based workflows

Whether you're at the REPL or in IJulia, you can typically improve your development experience with Revise. It is common to configure Revise to start whenever julia is started, as per the instructions in the Revise documentation. Once configured, Revise will track changes to files in any loaded modules, and to any files loaded in to the REPL with `includet` (but not with plain `include`); you can then edit the files and the changes take effect without restarting your julia session. A standard workflow is similar to the REPL-based workflow above, with the following modifications:

1. Put your code in a module somewhere on your load path. There are several options for achieving this, of which two recommended choices are:

   a. For long-term projects, use PkgTemplates:

   `julia using PkgTemplates t = Template() generate("MyPkg", t)` This will create a blank package, `"MyPkg"`, in your `.julia/dev` directory. Note that PkgTemplates allows you to control many different options through its `Template` constructor.

   In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and `MyPkg/test/runtests.jl` for the tests.

   b. For "throw-away" projects, you can avoid any need for cleanup by doing your work in your temporary directory (e.g., `/tmp`).

   Navigate to your temporary directory and launch Julia, then do the following:

   `julia pkg> generate MyPkg # type ] to enter pkg mode julia> push!(LOAD_PATH, pwd()) # hit backspace to exit pkg mode` If you restart your Julia session you'll have to re-issue that command modifying `LOAD_PATH`.

   In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and create any test file of your choosing.

2. Develop your package

   Before loading any code, make sure you're running Revise: say `using Revise` or follow its documentation on configuring it to run automatically.

   Then navigate to the directory containing your test file (here assumed to be `"runtests.jl"`) and do the following:

   ```
   julia> using MyPkg


   julia> include("runtests.jl")
   ```

   You can iteratively modify the code in MyPkg in your editor and re-run the tests with `include("runtests.jl")`. You generally should not need to restart your Julia session to see the changes take effect (subject to a few limitations, see https://timholy.github.io/Revise.jl/stable/limitations/).

Chapter 41

Style Guide

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

41.1 Write functions, not just scripts

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like `pi`).

41.2 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
Complex{Float64}(x)
```

it's better to use available generic functions:

```
complex(float(x))
```

The second version will convert `x` to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type `Int` or `Int32` if it really could be any integer, expressed with the abstract type `Integer`. In fact, in many cases you

can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a `MethodError` will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as duck typing.)

For example, consider the following definitions of a function `addone` that returns one plus its argument:

```
addone(x::Int) = x + 1                # works only for Int
addone(x::Integer) = x + oneunit(x)   # any integer type
addone(x::Number) = x + oneunit(x)    # any numeric type
addone(x) = x + oneunit(x)            # any type supporting + and oneunit
```

The last definition of `addone` handles any type supporting `oneunit` (which returns 1 in the same type as `x`, which avoids unwanted type promotion) and the `+` function with those arguments. The key thing to realize is that there is no performance penalty to defining only the general `addone(x) = x + oneunit(x)`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `addone(12)`, Julia will automatically compile a specialized `addone` function for `x::Int` arguments, with the call to `oneunit` replaced by its inlined value `1`. Therefore, the first three definitions of `addone` above are completely redundant with the fourth definition.

41.3 Handle excess argument diversity in the caller

Instead of:

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

This is better style because `foo` does not really accept numbers of all types; it really needs `Int` s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more "space" for future method definitions.

41.4 Append ! to names of functions that modify their arguments

Instead of:

```
function double(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

use:

```
function double!(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

Julia Base uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `sort` and `sort!`), and others which are just modifying (e.g., `push!`, `pop!`, `splice!`). It is typical for such functions to also return the modified array for convenience.

41.5 Avoid strange type `Union`s

Types such as `Union{Function,AbstractString}` are often a sign that some design could be cleaner.

41.6 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
a = Vector{Union{Int,AbstractString,Tuple,Array}}(undef, n)
```

In this case `Vector{Any}(undef, n)` is better. It is also more helpful to the compiler to annotate specific uses (e.g. `a[i]::Int`) than to try to pack many alternatives into one type.

41.7 Use naming conventions consistent with Julia `base/`

- modules and type names use capitalization and camel case: `module SparseArrays`, `struct UnitRange`.
- functions are lowercase (`maximum`, `convert`) and, when readable, with multiple words squashed together (`isequal`, `haskey`). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (`remotecall_fetch` as a more efficient implementation of `fetch(remotecall(...))`) or as modifiers.
- conciseness is valued, but avoid abbreviation (`indexin` rather than `indxin`) as it becomes difficult to remember whether and how particular words are abbreviated.

If a function name requires multiple words, consider whether it might represent more than one concept and might be better split into pieces.

41.8 Write functions with argument ordering similar to Julia Base

As a general rule, the Base library uses the following order of arguments to functions, as applicable:

1. Function argument. Putting a function argument first permits the use of `do` blocks for passing multiline anonymous functions.
2. I/O stream. Specifying the `IO` object first permits passing the function to functions such as `sprint`, e.g. `sprint(show, x)`.
3. Input being mutated. For example, in `fill!(x, v)`, `x` is the object being mutated and it appears before the value to be inserted into `x`.
4. Type. Passing a type typically means that the output will have the given type. In `parse(Int, "1")`, the type comes before the string to parse. There are many such examples where the type appears first, but it's useful to note that in `read(io, String)`, the `IO` argument appears before the type, which is in keeping with the order outlined here.
5. Input not being mutated. In `fill!(x, v)`, `v` is not being mutated and it comes after `x`.
6. Key. For associative collections, this is the key of the key-value pair(s). For other indexed collections, this is the index.
7. Value. For associative collections, this is the value of the key-value pair(s). In cases like `fill!(x, v)`, this is `v`.
8. Everything else. Any other arguments.

9. Varargs. This refers to arguments that can be listed indefinitely at the end of a function call. For example, in `Matrix{T}(undef, dims)`, the dimensions can be given as a `Tuple`, e.g. `Matrix{T}(undef, (1,2))`, or as `Varargs`, e.g. `Matrix{T}(undef, 1, 2)`.

10. Keyword arguments. In Julia keyword arguments have to come last anyway in function definitions; they're listed here for the sake of completeness.

The vast majority of functions will not take every kind of argument listed above; the numbers merely denote the precedence that should be used for any applicable arguments to a function.

There are of course a few exceptions. For example, in `convert`, the type should always come first. In `setindex!`, the value comes before the indices so that the indices can be provided as varargs.

When designing APIs, adhering to this general order as much as possible is likely to give users of your functions a more consistent experience.

41.9 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

41.10 Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

41.11 Don't overuse `...`

Splicing function arguments can be addictive. Instead of `[a..., b...]`, use simply `[a; b]`, which already concatenates arrays. `collect(a)` is better than `[a...]`, but since `a` is already iterable it is often even better to leave it alone, and not convert it to an array.

41.12 Don't use unnecessary static parameters

A function signature:

```
foo(x::T) where {T<:Real} = ...
```

should be written as:

```
foo(x::Real) = ...
```

instead, especially if `T` is not used in the function body. Even if `T` is used, it can be replaced with `typeof(x)` if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ Avoid fields with abstract containers for more information.

41.13 Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

Decide whether the concept in question will be written as `MyType` or `MyType()`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `Type{MyType}` later if they become necessary to solve some problem.

If a type is effectively an enumeration, it should be defined as a single (ideally immutable struct or primitive) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

41.14 Don't overuse macros

Be aware of when a macro could really be a function instead.

Calling `eval` inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

41.15 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
end
```

don't write definitions like the following:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

The problem is that users of this type can write `x[i]` without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have `unsafe` somewhere in its name to alert callers.

41.16 Don't overload methods of base container types

It is possible to write definitions like the following:

```
show(io::IO, v::Vector{MyType}) = ...
```

This would provide custom showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like `Vector()` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

41.17 Avoid type piracy

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In some cases, you can get away with type piracy with little ill effect. In extreme cases, however, you can even crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a `ccall`). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

As an example, suppose you wanted to define multiplication on symbols in a module:

```
module A
import Base.*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

The problem is that now any other module that uses `Base.*` will also see this definition. Since `Symbol` is defined in Base and is used by other modules, this can change the behavior of unrelated code unexpectedly. There are several alternatives here, including using a different function name, or wrapping the `Symbol`s in another type that you define.

Sometimes, coupled packages may engage in type piracy to separate features from definitions, especially when the packages were designed by collaborating authors, and when the definitions are reusable. For example, one package might provide some types useful for working with colors; another package could define methods for those types that enable conversions between color spaces. Another example might be a package that acts as a thin wrapper for some C code, which another package might then pirate to implement a higher-level, Julia-friendly API.

41.18 Be careful with type equality

You generally want to use `isa` and `<:` for testing types, not `==`. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `T == Float64`), or if you really, really know what you're doing.

41.19 Do not write `x->f(x)`

Since higher-order functions are often called with anonymous functions, it is easy to conclude that this is desirable or even necessary. But any function can be passed directly, without being "wrapped" in an anonymous function. Instead of writing `map(x->f(x), a)`, write `map(f, a)`.

41.20 Avoid using floats for numeric literals in generic code when possible

If you write generic code which handles numbers, and which can be expected to run with many different numeric type arguments, try using literals of a numeric type that will affect the arguments as little as possible through promotion.

For example,

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)

julia> f(1//2)
1.0
```

```
julia> f(1/2)
1.0

julia> f(1)
2.0
```

while

```
julia> g(x) = 2 * x
g (generic function with 1 method)

julia> g(1//2)
1//1

julia> g(1/2)
1.0

julia> g(1)
2
```

As you can see, the second version, where we used an `Int` literal, preserved the type of the input argument, while the first didn't. This is because e.g. `promote_type(Int, Float64) == Float64`, and promotion happens with the multiplication. Similarly, `Rational` literals are less type disruptive than `Float64` literals, but more disruptive than `Int`s:

```
julia> h(x) = 2//1 * x
h (generic function with 1 method)

julia> h(1//2)
1//1

julia> h(1/2)
1.0

julia> h(1)
2//1
```

Thus, use `Int` literals when possible, with `Rational{Int}` for literal non-integer numbers, in order to make it easier to use your code.

Chapter 42

Frequently Asked Questions

42.1 General

Is Julia named after someone or something?

No.

Why don't you compile Matlab/Python/R/⋯ code to Julia?

Since many people are familiar with the syntax of other dynamic languages, and lots of code has already been written in those languages, it is natural to wonder why we didn't just plug a Matlab or Python front-end into a Julia back-end (or "transpile" code to Julia) in order to get all the performance benefits of Julia without requiring programmers to learn a new language. Simple, right?

The basic issue is that there is nothing special about Julia's compiler: we use a commonplace compiler (LLVM) with no "secret sauce" that other language developers don't know about. Indeed, Julia's compiler is in many ways much simpler than those of other dynamic languages (e.g. PyPy or LuaJIT). Julia's performance advantage derives almost entirely from its front-end: its language semantics allow a well-written Julia program to give more opportunities to the compiler to generate efficient code and memory layouts. If you tried to compile Matlab or Python code to Julia, our compiler would be limited by the semantics of Matlab or Python to producing code no better than that of existing compilers for those languages (and probably worse). The key role of semantics is also why several existing Python compilers (like Numba and Pythran) only attempt to optimize a small subset of the language (e.g. operations on Numpy arrays and scalars), and for this subset they are already doing at least as well as we could for the same semantics. The people working on those projects are incredibly smart and have accomplished amazing things, but retrofitting a compiler onto a language that was designed to be interpreted is a very difficult problem.

Julia's advantage is that good performance is not limited to a small subset of "built-in" types and operations, and one can write high-level type-generic code that works on arbitrary user-defined types while remaining fast and

memory-efficient. Types in languages like Python simply don't provide enough information to the compiler for similar capabilities, so as soon as you used those languages as a Julia front-end you would be stuck.

For similar reasons, automated translation to Julia would also typically generate unreadable, slow, non-idiomatic code that would not be a good starting point for a native Julia port from another language.

On the other hand, language interoperability is extremely useful: we want to exploit existing high-quality code in other languages from Julia (and vice versa)! The best way to enable this is not a transpiler, but rather via easy inter-language calling facilities. We have worked hard on this, from the built-in `ccall` intrinsic (to call C and Fortran libraries) to JuliaInterop packages that connect Julia to Python, Matlab, C++, and more.

42.2 Sessions and the REPL

How do I delete an object in memory?

Julia does not have an analog of MATLAB's `clear` function; once a name is defined in a Julia session (technically, in module `Main`), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if `A` is a gigabyte-sized array that you no longer need, you can free the memory with `A = nothing`. The memory will be released the next time the garbage collector runs; you can force this to happen with `gc()`. Moreover, an attempt to use `A` will likely result in an error, because most methods are not defined on type `Nothing`.

How can I modify the declaration of a type in my session?

Perhaps you've defined a type and then realize you need to add a new field. If you try this at the REPL, you get the error:

```
ERROR: invalid redefinition of constant MyType
```

Types in module `Main` cannot be redefined.

While this can be inconvenient when you are developing new code, there's an excellent workaround. Modules can be replaced by redefining them, and so if you wrap all your new code inside a module you can redefine types and constants. You can't import the type names into `Main` and then expect to be able to redefine them there, but you can use the module name to resolve the scope. In other words, while developing you might use a workflow something like this:

```
include("mynewcode.jl")              # this defines a module MyModule
obj1 = MyModule.ObjConstructor(a, b)
obj2 = MyModule.somefunction(obj1)
```

```
# Got an error. Change something in "mynewcode.jl"
include("mynewcode.jl")              # reload the module
obj1 = MyModule.ObjConstructor(a, b) # old objects are no longer valid, must reconstruct
obj2 = MyModule.somefunction(obj1)   # this time it worked!
obj3 = MyModule.someotherfunction(obj2, c)
...
```

42.3 Scripting

How do I check if the current file is being run as the main script?

When a file is run as the main script using `julia file.jl` one might want to activate extra functionality like command line argument handling. A way to determine that a file is run in this fashion is to check if `abspath(PROGRAM_FILE) == @__FILE__` is `true`.

How do I catch CTRL-C in a script?

Running a Julia script using `julia file.jl` does not throw `InterruptException` when you try to terminate it with CTRL-C (SIGINT). To run a certain code before terminating a Julia script, which may or may not be caused by CTRL-C, use `atexit`. Alternatively, you can use `julia -e 'include(popfirst!(ARGS))' file.jl` to execute a script while being able to catch `InterruptException` in the `try` block.

How do I pass options to `julia` using `#!/usr/bin/env`?

Passing options to `julia` in so-called shebang by, e.g., `#!/usr/bin/env julia --startup-file=no` may not work in some platforms such as Linux. This is because argument parsing in shebang is platform-dependent and not well-specified. In a Unix-like environment, a reliable way to pass options to `julia` in an executable script would be to start the script as a `bash` script and use `exec` to replace the process to `julia`:

```
#!/bin/bash
#=
exec julia --color=yes --startup-file=no -e 'include(popfirst!(ARGS))' \
    "${BASH_SOURCE[0]}" "$@"
=#

@show ARGS  # put any Julia code here
```

In the example above, the code between `#=` and `=#` is run as a `bash` script. Julia ignores this part since it is a multi-line comment for Julia. The Julia code after `=#` is ignored by `bash` since it stops parsing the file once it reaches to the `exec` statement.

42.4 Functions

I passed an argument `x` to a function, modified it inside that function, but on the outside, the variable `x` is still unchanged. Why?

Suppose you call a function like this:

```
julia> x = 10
10

julia> function change_value!(y)
           y = 17
       end
change_value! (generic function with 1 method)

julia> change_value!(x)
17

julia> x # x is unchanged!
10
```

In Julia, the binding of a variable `x` cannot be changed by passing `x` as an argument to a function. When calling `change_value!(x)` in the above example, `y` is a newly created variable, bound initially to the value of `x`, i.e. `10`; then `y` is rebound to the constant `17`, while the variable `x` of the outer scope is left untouched.

But here is a thing you should pay attention to: suppose `x` is bound to an object of type `Array` (or any other mutable type). From within the function, you cannot "unbind" `x` from this Array, but you can change its content. For example:

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> function change_array!(A)
           A[1] = 5
       end
change_array! (generic function with 1 method)

julia> change_array!(x)
```

```
5


julia> x
3-element Array{Int64,1}:
 5
 2
 3
```

Here we created a function `change_array!`, that assigns `5` to the first element of the passed array (bound to `x` at the call site, and bound to `A` within the function). Notice that, after the function call, `x` is still bound to the same array, but the content of that array changed: the variables `A` and `x` were distinct bindings referring to the same mutable `Array` object.

Can I use `using` or `import` inside a function?

No, you are not allowed to have a `using` or `import` statement inside a function. If you want to import a module but only use its symbols inside a specific function or set of functions, you have two options:

1. Use `import`:

   ```
   import Foo
   function bar(...)
       # ... refer to Foo symbols via Foo.baz ...
   end
   ```

   This loads the module `Foo` and defines a variable `Foo` that refers to the module, but does not import any of the other symbols from the module into the current namespace. You refer to the `Foo` symbols by their qualified names `Foo.bar` etc.

2. Wrap your function in a module:

   ```
   module Bar
   export bar
   using Foo
   function bar(...)
       # ... refer to Foo.baz as simply baz ....
   end
   end
   using Bar
   ```

   This imports all the symbols from `Foo`, but only inside the module `Bar`.

What does the ... operator do?

The two uses of the ... operator: slurping and splatting

Many newcomers to Julia find the use of ... operator confusing. Part of what makes the ... operator confusing is that it means two different things depending on context.

... combines many arguments into one argument in function definitions

In the context of function definitions, the ... operator is used to combine many different arguments into a single argument. This use of ... for combining many different arguments into a single argument is called slurping:

```
julia> function printargs(args...)
           println(typeof(args))
           for (i, arg) in enumerate(args)
               println("Arg #$i = $arg")
           end
       end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
Tuple{Int64,Int64,Int64}
Arg #1 = 1
Arg #2 = 2
Arg #3 = 3
```

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as <-... instead of ....

... splits one argument into many different arguments in function calls

In contrast to the use of the ... operator to denote slurping many different arguments into one argument when defining a function, the ... operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of ... is called splatting:

```
julia> function threeargs(a, b, c)
           println("a = $a::$(typeof(a))")
           println("b = $b::$(typeof(b))")
           println("c = $c::$(typeof(c))")
       end
```

```
threeargs (generic function with 1 method)

julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> threeargs(x...)
a = 1::Int64
b = 2::Int64
c = 3::Int64
```

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as `...->` instead of `...`.

What is the return value of an assignment?

The operator `=` always returns the right-hand side, therefore:

```
julia> function threeint()
           x::Int = 3.0
           x # returns variable x
       end
threeint (generic function with 1 method)

julia> function threefloat()
           x::Int = 3.0 # returns 3.0
       end
threefloat (generic function with 1 method)

julia> threeint()
3

julia> threefloat()
3.0
```

and similarly:

```
julia> function threetup()
           x, y = [3, 3]
           x, y # returns a tuple
       end
threetup (generic function with 1 method)

julia> function threearr()
           x, y = [3, 3] # returns an array
       end
threearr (generic function with 1 method)

julia> threetup()
(3, 3)

julia> threearr()
2-element Array{Int64,1}:
 3
 3
```

42.5 Types, type declarations, and constructors

What does "type-stable" mean?

It means that the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the values of the inputs. The following code is not type-stable:

```
julia> function unstable(flag::Bool)
           if flag
               return 1
           else
               return 1.0
           end
       end
unstable (generic function with 1 method)
```

It returns either an `Int` or a `Float64` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it will have to guard against both types possibly occurring, making generation of fast machine code difficult.

Why does Julia give a `DomainError` for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

```
julia> sqrt(-2.0)
ERROR: DomainError with -2.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `sqrt`, most users want `sqrt(2.0)` to give a real number, and would be unhappy if it produced the complex number `1.4142135623730951 + 0.0im`. One could write the `sqrt` function to switch to a complex-valued output only when passed a negative number (which is what `sqrt` does in some other languages), but then the result would not be type-stable and the `sqrt` function would have poor performance.

In these and other cases, you can get the result you want by choosing an input type that conveys your willingness to accept an output type in which the result can be represented:

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> typemax(Int)
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `Int128` or `BigInt` in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) — it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, type-stability is crucial for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability is to merge the `Int` and `BigInt` types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach can be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots — large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps — situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1

ans =
```

```
  9223372036854775807

>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

 -9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow is associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
julia> n = 2^62
4611686018427387904
```

```
julia> (n + 2n) >>> 1
6917529027641081856
```

See? No problem. That's the correct midpoint between 2^62 and 2^63, despite the fact that `n + 2n` is -4611686018427387904. Now try it in Matlab:

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

Oops. Adding a `>>>` operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
julia> code_native(f, Tuple{Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 1
  leaq  -1(%rdi,%rdi,4), %rax
  popq  %rbp
  retq
  nopl  (%rax,%rax)
```

The actual body of the function is a single `leaq` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k, n)
           for i = 1:n
               k = f(k)
           end
           return k
       end
```

```
g (generic function with 1 methods)

julia> code_native(g, Tuple{Int,Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 2
  testq %rsi, %rsi
  jle L26
  nopl  (%rax)
Source line: 3
L16:
  leaq  -1(%rdi,%rdi,4), %rdi
Source line: 2
  decq  %rsi
  jne L16
Source line: 5
L26:
  movq  %rdi, %rax
  popq  %rbp
  retq
  nop
```

Since the call to `f` gets inlined, the loop body ends up being just a single `leaq` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
julia> function g(k)
           for i = 1:10
               k = f(k)
           end
           return k
       end
g (generic function with 2 methods)

julia> code_native(g,(Int,))
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
```

```
Source line: 3
  imulq $9765625, %rdi, %rax    # imm = 0x9502F9
  addq  $-2441406, %rax         # imm = 0xFFDABF42
Source line: 5
  popq  %rbp
  retq
  nopw  %cs:(%rax,%rax)
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

The most reasonable alternative to having integer arithmetic silently overflow is to do checked arithmetic everywhere, raising errors when adds, subtracts, and multiplies overflow, producing values that are not value-correct. In this blog post, Dan Luu analyzes this and finds that rather than the trivial cost that this approach should in theory have, it ends up having a substantial cost due to compilers (LLVM and GCC) not gracefully optimizing around the added overflow checks. If this improves in the future, we could consider defaulting to checked integer arithmetic in Julia, but for now, we have to live with the possibility of overflow.

What are the possible causes of an `UndefVarError` during remote execution?

As the error states, an immediate cause of an `UndefVarError` on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

```
julia> module Foo
           foo() = remotecall_fetch(x->x, 2, "Hello")
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: Foo not defined
Stacktrace:
[...]
```

The closure `x->x` carries a reference to `Foo`, and since `Foo` is unavailable on node 2, an `UndefVarError` is thrown.

Globals under modules other than `Main` are not serialized by value to the remote node. Only a reference is sent. Functions which create global bindings (except under `Main`) may cause an `UndefVarError` to be thrown later.

```
julia> @everywhere module Foo
           function foo()
               global gvar = "Hello"
               remotecall_fetch(()->gvar, 2)
           end
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: gvar not defined
Stacktrace:
[...]
```

In the above example, `@everywhere module Foo` defined `Foo` on all nodes. However the call to `Foo.foo()` created a new global binding `gvar` on the local node, but this was not found on node 2 resulting in an `UndefVarError` error.

Note that this does not apply to globals created under module `Main`. Globals under module `Main` are serialized and new bindings created under `Main` on the remote node.

```
julia> gvar_self = "Node1"
"Node1"

julia> remotecall_fetch(()->gvar_self, 2)
"Node1"

julia> remotecall_fetch(varinfo, 2)
name          size summary
_________ _______ _______
Base              Module
Core              Module
Main              Module
gvar_self 13 bytes String
```

This does not apply to `function` or `struct` declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

```
julia> bar() = 1
bar (generic function with 1 method)

julia> remotecall_fetch(bar, 2)
ERROR: On worker 2:
UndefVarError: #bar not defined
[...]

julia> anon_bar  = ()->1
(::#21) (generic function with 1 method)

julia> remotecall_fetch(anon_bar, 2)
1
```

Why does Julia use * for string concatenation? Why not + or something else?

The main argument against + is that string concatenation is not commutative, while + is generally used as a commutative operator. While the Julia community recognizes that other languages use different operators and * may be unfamiliar for some users, it communicates certain algebraic properties.

Note that you can also use `string(...)` to concatenate strings (and other values converted to strings); similarly, `repeat` can be used instead of ^ to repeat strings. The interpolation syntax is also useful for constructing strings.

42.6 Packages and Modules

What is the difference between "using" and "import"?

There is only one difference, and on the surface (syntax-wise) it may seem very minor. The difference between `using` and `import` is that with `using` you need to say `function Foo.bar(..` to extend module Foo's function bar with a new method, but with `import Foo.bar`, you only need to say `function bar(...` and it automatically extends module Foo's function bar.

The reason this is important enough to have been given separate syntax is that you don't want to accidentally extend a function that you didn't know existed, because that could easily cause a bug. This is most likely to happen with a method that takes a common type like a string or integer, because both you and the other module could define a method to handle such a common type. If you use `import`, then you'll replace the other module's implementation of `bar(s::AbstractString)` with your new implementation, which could easily do something completely different (and break all/many future usages of the other functions in module Foo that depend on calling bar).

42.7 Nothingness and missing values

How does "null", "nothingness" or "missingness" work in Julia?

Unlike many languages (for example, C and Java), Julia objects cannot be "null" by default. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the `isdefined` or `isassigned` functions.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value `nothing`, which is just a singleton object of type `Nothing`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield `nothing`, for example `if false; end`.

For situations where a value `x` of type `T` exists only sometimes, the `Union{T, Nothing}` type can be used for function arguments, object fields and array element types as the equivalent of `Nullable`, `Option` or `Maybe` in other languages. If the value itself can be `nothing` (notably, when `T` is `Any`), the `Union{Some{T}, Nothing}` type is more appropriate since `x == nothing` then indicates the absence of a value, and `x == Some(nothing)` indicates the presence of a value equal to `nothing`. The `something` function allows unwrapping `Some` objects and using a default value instead of `nothing` arguments. Note that the compiler is able to generate efficient code when working with `Union{T, Nothing}` arguments or fields.

To represent missing data in the statistical sense (`NA` in R or `NULL` in SQL), use the `missing` object. See the `Missing Values` section for more details.

The empty tuple (`()`) is another form of nothingness. But, it should not really be thought of as nothing but rather a tuple of zero values.

The empty (or "bottom") type, written as `Union{}` (an empty union type), is a type with no values and no subtypes (except itself). You will generally not need to use this type.

42.8 Memory

Why does `x += y` allocate memory when `x` and `y` are arrays?

In Julia, `x += y` gets replaced during parsing by `x = x + y`. For arrays, this has the consequence that, rather than storing the result in the same location in memory as `x`, it allocates a new array to store the result.

While this behavior might surprise some, the choice is deliberate. The main reason is the presence of immutable objects within Julia, which cannot change their value once created. Indeed, a number is an immutable object; the statements `x = 5; x += 1` do not modify the meaning of `5`, they modify the value bound to `x`. For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

```
function power_by_squaring(x, n::Int)
    ispow2(n) || error("This implementation only works for powers of 2")
    while n >= 2
        x *= x
        n >>= 1
    end
    x
end
```

After a call like `x = 5; y = power_by_squaring(x, 4)`, you would get the expected result: `x == 5 && y == 625`. However, now suppose that `*=`, when used with matrices, instead mutated the left hand side. There would be two problems:

- For general square matrices, `A = A*B` cannot be implemented without temporary storage: `A[1,1]` gets computed and stored on the left hand side before you're done using it on the right hand side.
- Suppose you were willing to allocate a temporary for the computation (which would eliminate most of the point of making `*=` work in-place); if you took advantage of the mutability of `x`, then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable `x`, after the call you'd have (in general) `y != x`, but for mutable `x` you'd have `y == x`.

Because supporting generic programming is deemed more important than potential performance optimizations that can be achieved by other means (e.g., using explicit loops), operators like `+=` and `*=` work by rebinding new values.

42.9 Asynchronous IO and concurrent synchronous writes

Why do concurrent writes to the same stream result in inter-mixed output?

While the streaming I/O API is synchronous, the underlying implementation is fully asynchronous.

Consider the printed output from the following:

```
julia> @sync for i in 1:3
           @async write(stdout, string(i), " Foo ", " Bar ")
       end
123 Foo  Foo  Foo  Bar  Bar  Bar
```

This is happening because, while the `write` call is synchronous, the writing of each argument yields to other tasks while waiting for that part of the I/O to complete.

`print` and `println` "lock" the stream during a call. Consequently changing `write` to `println` in the above example results in:

```
julia> @sync for i in 1:3
           @async println(stdout, string(i), " Foo ", " Bar ")
       end
1 Foo  Bar
2 Foo  Bar
3 Foo  Bar
```

You can lock your writes with a `ReentrantLock` like this:

```
julia> l = ReentrantLock();

julia> @sync for i in 1:3
           @async begin
               lock(l)
               try
                   write(stdout, string(i), " Foo ", " Bar ")
               finally
                   unlock(l)
               end
           end
       end
1 Foo  Bar 2 Foo  Bar 3 Foo  Bar
```

42.10 Arrays

What are the differences between zero-dimensional arrays and scalars?

Zero-dimensional arrays are arrays of the form `Array{T,0}`. They behave similar to scalars, but there are important differences. They deserve a special mention because they are a special case which makes logical sense given the generic definition of arrays, but might be a bit unintuitive at first. The following line defines a zero-dimensional array:

```
julia> A = zeros()
0-dimensional Array{Float64,0}:
0.0
```

In this example, `A` is a mutable container that contains one element, which can be set by `A[] = 1.0` and retrieved with `A[]`. All zero-dimensional arrays have the same size (`size(A) == ()`), and length (`length(A) == 1`). In particular, zero-dimensional arrays are not empty. If you find this unintuitive, here are some ideas that might help to understand Julia's definition.

- Zero-dimensional arrays are the "point" to vector's "line" and matrix's "plane". Just as a line has no area (but still represents a set of things), a point has no length or any dimensions at all (but still represents a thing).
- We define `prod(())` to be 1, and the total number of elements in an array is the product of the size. The size of a zero-dimensional array is `()`, and therefore its length is `1`.
- Zero-dimensional arrays don't natively have any dimensions into which you index – they're just `A[]`. We can apply the same "trailing one" rule for them as for all other array dimensionalities, so you can indeed index them as `A[1]`, `A[1,1]`, etc; see Omitted and extra indices.

It is also important to understand the differences to ordinary scalars. Scalars are not mutable containers (even though they are iterable and define things like `length`, `getindex`, e.g. `1[] == 1`). In particular, if `x = 0.0` is defined as a scalar, it is an error to attempt to change its value via `x[] = 1.0`. A scalar `x` can be converted into a zero-dimensional array containing it via `fill(x)`, and conversely, a zero-dimensional array `a` can be converted to the contained scalar via `a[]`. Another difference is that a scalar can participate in linear algebra operations such as `2 * rand(2,2)`, but the analogous operation with a zero-dimensional array `fill(2) * rand(2,2)` is an error.

Why are my Julia benchmarks for linear algebra operations different from other languages?

You may find that simple benchmarks of linear algebra building blocks like

```
using BenchmarkTools
A = randn(1000, 1000)
B = randn(1000, 1000)
@btime $A \ $B
@btime $A * $B
```

can be different when compared to other languages like Matlab or R.

Since operations like this are very thin wrappers over the relevant BLAS functions, the reason for the discrepancy is very likely to be

1. the BLAS library each language is using,
2. the number of concurrent threads.

Julia compiles and uses its own copy of OpenBLAS, with threads currently capped at 8 (or the number of your cores).

Modifying OpenBLAS settings or compiling Julia with a different BLAS library, eg Intel MKL, may provide performance improvements. You can use MKL.jl, a package that makes Julia's linear algebra use Intel MKL BLAS and LAPACK instead of OpenBLAS, or search the discussion forum for suggestions on how to set this up manually. Note that Intel MKL cannot be bundled with Julia, as it is not open source.

42.11 Julia Releases

Do I want to use the Stable, LTS, or nightly version of Julia?

The Stable version of Julia is the latest released version of Julia, this is the version most people will want to run. It has the latest features, including improved performance. The Stable version of Julia is versioned according to SemVer as v1.x.y. A new minor release of Julia corresponding to a new Stable version is made approximately every 4-5 months after a few weeks of testing as a release candidate. Unlike the LTS version the a Stable version will not normally recieve bugfixes after another Stable version of Julia has been released. However, upgrading to the next Stable release will always be possible as each release of Julia v1.x will continue to run code written for earlier versions.

You may prefer the LTS (Long Term Support) version of Julia if you are looking for a very stable code base. The current LTS version of Julia is versioned according to SemVer as v1.0.x; this branch will continue to recieve bugfixes until a new LTS branch is chosen, at which point the v1.0.x series will no longer recieved regular bug fixes and all but the most conservative users will be advised to upgrade to the new LTS version series. As a package developer, you may prefer to develop for the LTS version, to maximize the number of users who can use your package. As per SemVer, code written for v1.0 will continue to work for all future LTS and Stable versions. In general, even if targetting the LTS, one can develop and run code in the latest Stable version, to take advantage of the improved performance; so long as one avoids using new features (such as added library functions or new methods).

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work. As the name implies, releases to the nightly version are made roughly every night (depending on build infrastructure stability). In general nightly released are fairly safe to use—your code will not catch on fire. However, they may be occasional regressions and or issues that will not be found until more thorough pre-release testing. You may wish to test against the nightly version to ensure that such regressions that affect your use case are caught before a release is made.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our guidelines for contributing.

Links to each of these download types can be found on the download page at https://julialang.org/downloads/. Note that not all versions of Julia are available for all platforms.

Chapter 43

Noteworthy Differences from other Languages

43.1 Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is not a MATLAB clone. There are major syntactic and functional differences. The following are some noteworthy differences that may trip up Julia users accustomed to MATLAB:

- Julia arrays are indexed with square brackets, `A[i,j]`.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well.
- Julia values are not copied when passed to a function. If a function modifies an array, the changes will be visible in the caller.
- Julia does not automatically grow arrays in an assignment statement. Whereas in MATLAB `a(4) = 3.2` can create the array `a = [0 0 0 3.2]` and `a(5) = 7` can grow it into `a = [0 0 0 3.2 7]`, the corresponding Julia statement `a[5] = 7` throws an error if the length of `a` is less than 5 or if this statement is the first use of the identifier `a`. Julia has `push!` and `append!`, which grow `Vector`s much more efficiently than MATLAB's `a(end+1) = val`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `i` or `j` as in MATLAB.
- In Julia, literal numbers without a decimal point (such as `42`) create integers instead of floating point numbers. As a result, some operations can throw a domain error if they expect a float; for example, `julia> a = -1; 2^a` throws a domain error, as the result is not an integer (see the FAQ entry on domain errors for details).
- In Julia, multiple values are returned and assigned as tuples, e.g. `(a, b) = (1, 2)` or `a, b = 1, 2`. MATLAB's `nargout`, which is often used in MATLAB to do optional work based on the number of returned values, does not exist in Julia. Instead, users can use optional and keyword arguments to achieve similar capabilities.

- Julia has true one-dimensional arrays. Column vectors are of size `N`, not `Nx1`. For example, `rand(N)` makes a 1-dimensional array.
- In Julia, `[x,y,z]` will always construct a 3-element array containing `x`, `y` and `z`.
  - To concatenate in the first ("vertical") dimension use either `vcat(x,y,z)` or separate with semicolons (`[x; y; z]`).
  - To concatenate in the second ("horizontal") dimension use either `hcat(x,y,z)` or separate with spaces (`[x y z]`).
  - To construct block matrices (concatenating in the first two dimensions), use either `hvcat` or combine spaces and semicolons (`[a b; c d]`).
- In Julia, `a:b` and `a:b:c` construct `AbstractRange` objects. To construct a full vector like in MATLAB, use `collect(a:b)`. Generally, there is no need to call `collect` though. An `AbstractRange` object will act like a normal array in most cases but is more efficient because it lazily computes its values. This pattern of creating specialized objects instead of full arrays is used frequently, and is also seen in functions such as `range`, or with iterators such as `enumerate`, and `zip`. The special objects can mostly be used as if they were normal arrays.
- Functions in Julia return values from their last expression or the `return` keyword instead of listing the names of variables to return in the function definition (see return 키워드 for details).
- A Julia script may contain any number of functions, and all definitions will be externally visible when the file is loaded. Function definitions can be loaded from files outside the current working directory.
- In Julia, reductions such as `sum`, `prod`, and `max` are performed over every element of an array when called with a single argument, as in `sum(A)`, even if `A` has more than one dimension.
- In Julia, parentheses must be used to call a function with zero arguments, like in `rand()`.
- Julia discourages the use of semicolons to end statements. The results of statements are not automatically printed (except at the interactive prompt), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output.
- In Julia, if `A` and `B` are arrays, logical comparison operations like `A == B` do not return an array of booleans. Instead, use `A .== B`, and similarly for the other boolean operators like `<`, `>`.
- In Julia, the operators `&`, `¦`, and `⊻` (`xor`) perform the bitwise operations equivalent to `and`, `or`, and `xor` respectively in MATLAB, and have precedence similar to Python's bitwise operators (unlike C). They can operate on scalars or element-wise across arrays and can be used to combine logical arrays, but note the difference in order of operations: parentheses may be required (e.g., to select elements of `A` equal to 1 or 2 use `(A .== 1) .¦ (A .== 2)`).

- In Julia, the elements of a collection can be passed as arguments to a function using the splat operator `...`, as in `xs=[1,2]; f(xs...)`.
- Julia's `svd` returns singular values as a vector instead of as a dense diagonal matrix.
- In Julia, `...` is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.
- In both Julia and MATLAB, the variable `ans` is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, `ans` is not set when Julia code is run in non-interactive mode.
- Julia's `struct`s do not support dynamically adding fields at runtime, unlike MATLAB's `class`es. Instead, use a `Dict`.
- In Julia each module has its own global scope/namespace, whereas in MATLAB there is just one global scope.
- In MATLAB, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x(x>3)` or in the statement `x(x>3) = []` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.
- The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in `vertcat(A{:})` in MATLAB, is written using the splat operator in Julia, e.g. as `vcat(A...)`.

43.2 Noteworthy differences from R

One of Julia's goals is to provide an effective language for data analysis and statistical programming. For users coming to Julia from R, these are some noteworthy differences:

- Julia's single quotes enclose characters, not strings.
- Julia can create substrings by indexing into strings. In R, strings must be converted into character vectors before creating substrings.
- In Julia, like Python but unlike R, strings can be created with triple quotes `""" ... """`. This syntax is convenient for constructing strings that contain line breaks.
- In Julia, varargs are specified using the splat operator `...`, which always follows the name of a specific variable, unlike R, for which `...` can occur in isolation.
- In Julia, modulus is `mod(a, b)`, not `a %% b`. `%` in Julia is the remainder operator.

- In Julia, not all data structures support logical indexing. Furthermore, logical indexing in Julia is supported only with vectors of length equal to the object being indexed. For example:
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE)]` is equivalent to `c(1, 3)`.
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE, TRUE, FALSE)]` is equivalent to `c(1, 3)`.
  - In Julia, `[1, 2, 3, 4][[true, false]]` throws a `BoundsError`.
  - In Julia, `[1, 2, 3, 4][[true, false, true, false]]` produces `[1, 3]`.
- Like many languages, Julia does not always allow operations on vectors of different lengths, unlike R where the vectors only need to share a common index range. For example, `c(1, 2, 3, 4) + c(1, 2)` is valid R but the equivalent `[1, 2, 3, 4] + [1, 2]` will throw an error in Julia.
- Julia allows an optional trailing comma when that comma does not change the meaning of code. This can cause confusion among R users when indexing into arrays. For example, `x[1,]` in R would return the first row of a matrix; in Julia, however, the comma is ignored, so `x[1,] == x[1]`, and will return the first element. To extract a row, be sure to use `:`, as in `x[1,:]`.
- Julia's `map` takes the function first, then its arguments, unlike `lapply(<structure>, function, ...)` in R. Similarly Julia's equivalent of `apply(X, MARGIN, FUN, ...)` in R is `mapslices` where the function is the first argument.
- Multivariate apply in R, e.g. `mapply(choose, 11:13, 1:3)`, can be written as `broadcast(binomial, 11:13, 1:3)` in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions `binomial.(11:13, 1:3)`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`.
- In Julia, `<-`, `<<-` and `->` are not assignment operators.
- Julia's `->` creates an anonymous function.
- Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.
- Julia's `*` operator can perform matrix multiplication, unlike in R. If `A` and `B` are matrices, then `A * B` denotes a matrix multiplication in Julia, equivalent to R's `A %*% B`. In R, this same notation would perform an element-wise (Hadamard) product. To get the element-wise multiplication operation, you need to write `A .* B` in Julia.
- Julia performs matrix transposition using the `transpose` function and conjugated transposition using the `'` operator or the `adjoint` function. Julia's `transpose(A)` is therefore equivalent to R's `t(A)`. Additionally a non-recursive transpose in Julia is provided by the `permutedims` function.

- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (i in c(1, 2, 3))` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia does not provide `nrow` and `ncol`. Instead, use `size(M, 1)` for `nrow(M)` and `size(M, 2)` for `ncol(M)`.
- Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `c(1)` are the same. In Julia, they cannot be used interchangeably.
- Julia's `diag` and `diagm` are not like R's.
- Julia cannot assign to the results of function calls on the left hand side of an assignment operation: you cannot write `diag(M) = fill(1, n)`.
- Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in packages under the JuliaStats organization. For example:
  - Functions pertaining to probability distributions are provided by the Distributions package.
  - The DataFrames package provides data frames.
  - Generalized linear models are provided by the GLM package.
- Julia provides tuples and real hash tables, but not R-style lists. When returning multiple items, you should typically use a tuple or a named tuple: instead of `list(a = 1, b = 2)`, use `(1, 2)` or `(a=1, b=2)`.
- Julia encourages users to write their own types, which are easier to use than S3 or S4 objects in R. Julia's multiple dispatch system means that `table(x::TypeA)` and `table(x::TypeB)` act like R's `table.TypeA(x)` and `table.TypeB(x)`.
- In Julia, values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller. This is very different from R and allows new functions to operate on large data structures much more efficiently.
- In Julia, vectors and matrices are concatenated using `hcat`, `vcat` and `hvcat`, not `c`, `rbind` and `cbind` like in R.
- In Julia, a range like `a:b` is not shorthand for a vector like in R, but is a specialized `AbstractRange` object that is used for iteration without high memory overhead. To convert a range into a vector, use `collect(a:b)`.
- Julia's `max` and `min` are the equivalent of `pmax` and `pmin` respectively in R, but both arguments need to have the same dimensions. While `maximum` and `minimum` replace `max` and `min` in R, there are important differences.

- Julia's `sum`, `prod`, `maximum`, and `minimum` are different from their counterparts in R. They all accept an optional keyword argument `dims`, which indicates the dimensions, over which the operation is carried out. For instance, let `A = [1 2; 3 4]` in Julia and `B <- rbind(c(1,2),c(3,4))` be the same matrix in R. Then `sum(A)` gives the same result as `sum(B)`, but `sum(A, dims=1)` is a row vector containing the sum over each column and `sum(A, dims=2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where separate `colSums(B)` and `rowSums(B)` functions provide these functionalities. If the `dims` keyword argument is a vector, then it specifies all the dimensions over which the sum is performed, while retaining the dimensions of the summed array, e.g. `sum(A, dims=(1,2)) == hcat(10)`. It should be noted that there is no error checking regarding the second argument.
- Julia has several functions that can mutate their arguments. For example, it has both `sort` and `sort!`.
- In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.
- Julia is eagerly evaluated and does not support R-style lazy evaluation. For most users, this means that there are very few unquoted expressions or column names.
- Julia does not support the `NULL` type. The closest equivalent is `nothing`, but it behaves like a scalar value rather than like a list. Use `x === nothing` instead of `is.null(x)`.
- In Julia, missing values are represented by the `missing` object rather than by `NA`. Use `ismissing(x)` (or `ismissing.(x)` for element-wise operation on vectors) instead of `is.na(x)`. The `skipmissing` function is generally used instead of `na.rm=TRUE` (though in some particular cases functions take a `skipmissing` argument).
- Julia lacks the equivalent of R's `assign` or `get`.
- In Julia, `return` does not require parentheses.
- In R, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x[x>3]` or in the statement `x = x[x>3]` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.

43.3 Noteworthy differences from Python

- Julia requires `end` to end a block. Unlike Python, Julia has no `pass` keyword.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia's slice indexing includes the last element, unlike in Python. `a[2:3]` in Julia is `a[1:3]` in Python.

- Julia does not support negative indices. In particular, the last element of a list or array is indexed with `end` in Julia, not `-1` as in Python.
- Julia's `for`, `if`, `while`, etc. blocks are terminated by the `end` keyword. Indentation level is not significant as it is in Python.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of Performance Tips).
- Julia's updating operators (e.g. `+=`, `-=`, ...) are not in-place whereas NumPy's are. This means `A = [1, 1]; B = A; B += [3, 3]` doesn't change values in `A`, it rather rebinds the name `B` to the result of the right-hand side `B = B + 3`, which is a new array. For in-place operation, use `B .+= 3` (see also dot operators), explicit loops, or `InplaceOps.jl`.
- Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand, the function `g(x=[1,2]) = push!(x,3)` returns `[1,2,3]` every time it is called as `g()`.
- In Julia `%` is the remainder operator, whereas in Python it is the modulus.
- The commonly used `Int` type corresponds to the machine integer type (`Int32` or `Int64`). This means it will overflow, such that `2^64 == 0`. If you need larger values use another appropriate type, such as `Int128`, `BigInt` or a floating point type like `Float64`.

43.4 Noteworthy differences from C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension `A[i,j]`. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the Julia documentation for the syntax for array construction (it has changed between versions).
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `A = A + B` which rebinds the left-hand side to the result of the right-hand side expression.

- Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++ (see relevant section of Performance Tips).
- Julia values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller.
- In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.
- In Julia, literal numbers without a decimal point (such as `42`) create signed integers, of type `Int`, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as `Int64` (if `Int` is `Int32`), `Int128`, or the arbitrarily large `BigInt` type. There are no numeric literal suffixes, such as `L`, `LL`, `U`, `UL`, `ULL` to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with `0x` like C/C++), are unsigned. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the length of the literal, including leading 0s. For example, `0x0` and `0x00` have type `UInt8`, `0x000` and `0x0000` have type `UInt16`, then literals with 5 to 8 hex digits have type `UInt32`, 9 to 16 hex digits type `UInt64` and 17 to 32 hex digits type `UInt128`. This needs to be taken into account when defining hexadecimal masks, for example `~0xf == 0xf0` is very different from `~0x000f == 0xfff0`. 64 bit `Float64` and 32 bit `Float32` bit literals are expressed as `1.0` and `1.0f0` respectively. Floating point literals are rounded (and not promoted to the `BigFloat` type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with `0o`) and binary (prefixed with `0b`) literals are also treated as unsigned.
- String literals can be delimited with either " or """, """ delimited literals can contain " characters without quoting it like "\"". String literals can have values of other variables or expressions interpolated into them, indicated by `$variablename` or `$(expression)`, which evaluates the variable name or the expression in the context of the function.
- `//` indicates a `Rational` number, and not a single-line comment (which is # in Julia)
- `#=` indicates the start of a multiline comment, and `=#` ends it.
- Functions in Julia return values from their last expression(s) or the `return` keyword. Multiple values can be returned from functions and assigned as tuples, e.g. `(a, b) = myfunction()` or `a, b = myfunction()`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `a = myfunction(&b)`.
- Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output. In the REPL, `;` can be used to suppress output. `;` also has a different meaning within `[ ]`, something to watch out for. `;` can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.

- In Julia, the operator `⊻` (`xor`) performs the bitwise XOR operation, i.e. `^` in C/C++. Also, the bitwise operators do not have the same precedence as C/++, so parenthesis may be required.
- Julia's `^` is exponentiation (pow), not bitwise XOR as in C/C++ (use `⊻`, or `xor`, in Julia)
- Julia has two right-shift operators, `>>` and `>>>`. `>>>` performs an arithmetic shift, `>>` always performs a logical shift, unlike C/C++, where the meaning of `>>` depends on the type of the value being shifted.
- Julia's `->` creates an anonymous function, it does not access a member via a pointer.
- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (int i=1; i <= 3; i++)` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`, because of the operator precedence.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the `@` character, and have both a function-like syntax, `@mymacro(arg1, arg2, arg3)`, and a statement-like syntax, `@mymacro arg1 arg2 arg3`. The forms are interchangeable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the distributed `for` construct: `@distributed for i in 1:n; #= body =#; end`. Where the end of the macro construct may be unclear, use the function-like form.
- Julia has an enumeration type, expressed using the macro `@enum(name, value1, value2, ...)` For example: `@enum(Fruit, banana=1, apple, pear)`
- By convention, functions that modify their arguments have a `!` at the end of the name, for example `push!`.
- In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only `this`, using the most-specific-declaration rule).

43.5 Noteworthy differences from Common Lisp

- Julia uses 1-based indexing for arrays by default, and it can also handle arbitrary index offsets.
- Functions and variables share the same namespace ("Lisp-1").
- There is a `Pair` type, but it is not meant to be used as a `COMMON-LISP:CONS`. Various iterable collections can be used interchangeably in most parts of the language (eg splatting, tuples, etc). `Tuple`s are the closest to Common Lisp lists for short collections of heterogeneous elements. Use `NamedTuple`s in place of alists. For larger collections of homogeneous types, `Array`s and `Dict`s should be used.
- The typical Julia workflow for prototyping also uses continuous manipulation of the image, implemented with the Revise.jl package.
- Bignums are supported, but conversion is not automatic; ordinary integers overflow.
- Modules (namespaces) can be hierarchical. `import` and `using` have a dual role: they load the code and make it available in the namespace. `import` for only the module name is possible (roughly equivalent to `ASDF:LOAD-OP`). Slot names don't need to be exported separately. Global variables can't be assigned to from outside the module (except with `eval(mod, :(var = val))` as an escape hatch).
- Macros start with `@`, and are not as seamlessly integrated into the language as Common Lisp; consequently, macro usage is not as widespread as in the latter. A form of hygiene for macros is supported by the language. Because of the different surface syntax, there is no equivalent to `COMMON-LISP:&BODY`.
- All functions are generic and use multiple dispatch. Argument lists don't have to follow the same template, which leads to a powerful idiom (see `do`). Optional and keyword arguments are handled differently. Method ambiguities are not resolved like in the Common Lisp Object System, necessitating the definition of a more specific method for the intersection.
- Symbols do not belong to any package, and do not contain any values per se. `M.var` evaluates the symbol `var` in the module `M`.
- A functional programming style is fully supported by the language, including closures, but isn't always the idiomatic solution for Julia. Some workarounds may be necessary for performance when modifying captured variables.

Chapter 44

Unicode Input

The following table lists Unicode characters that can be entered via tab completion of LaTeX-like abbreviations in the Julia REPL (and in various other editing environments). You can also get information on how to type a symbol by entering it in the REPL help, i.e. by typing **?** and then entering the symbol in the REPL (e.g., by copy-paste from somewhere you saw the symbol).

> Warning
>
> This table may appear to contain missing characters in the second column, or even show characters that are inconsistent with the characters as they are rendered in the Julia REPL. In these cases, users are strongly advised to check their choice of fonts in their browser and REPL environment, as there are known issues with glyphs in many fonts.

Code point(s)	Char-ac-ter(s)	Tab completion sequence(s)	Unicode name(s)
U+000A1	¡	₩exclamdown	Inverted Exclamation Mark
U+000A3	&	₩sterling	Pound Sign
U+000A5	¥	₩yen	Yen Sign
U+000A6	☒	₩brokenbar	Broken Bar / Broken Vertical Bar
U+000A7	§	₩S	Section Sign
U+000A9	©	₩copyright, ₩:copyright:	Copyright Sign
U+000AA	ª	₩ordfeminine	Feminine Ordinal Indicator
U+000AC	¬	₩neg	Not Sign
U+000AE	☒	₩circledR, ₩:registered:	Registered Sign / Registered Trade Mark Sign
U+000AF	☒	₩highminus	Macron / Spacing Macron
U+000B0	⁰	₩degree	Degree Sign
U+000B1	±	₩pm	Plus-Minus Sign / Plus-Or-Minus Sign
U+000B2	²	₩^2	Superscript Two / Superscript Digit Two
U+000B3	³	₩^3	Superscript Three / Superscript Digit Three
U+000B6	¶	₩P	Pilcrow Sign / Paragraph Sign
U+000B7	·	₩cdotp	Middle Dot
U+000B9	¹	₩^1	Superscript One / Superscript Digit One
U+000BA	º	₩ordmasculine	Masculine Ordinal Indicator
U+000BC	¼	₩1/4	Vulgar Fraction One Quarter / Fraction One Quarter
U+000BD	½	₩1/2	Vulgar Fraction One Half / Fraction One Half
U+000BE	¾	₩3/4	Vulgar Fraction Three Quarters / Fraction Three Quarters
U+000BF	¿	₩questiondown	Inverted Question Mark
U+000C5	☒	₩AA	Latin Capital Letter A With Ring Above / Latin Capital Letter A Ring
U+000C6	Æ	₩AE	Latin Capital Letter Ae / Latin Capital Letter A E
U+000D0	Ð	₩DH	Latin Capital Letter Eth
U+000D7	×	₩times	Multiplication Sign
U+000D8	∅	₩O	Latin Capital Letter O With Stroke / Latin Capital Letter O Slash
U+000DE	Þ	₩TH	Latin Capital Letter Thorn
U+000DF	ß	₩ss	Latin Small Letter Sharp S
U+000E5	☒	₩aa	Latin Small Letter A With Ring Above / Latin Small Letter A Ring
U+000E6	æ	₩ae	Latin Small Letter Ae / Latin Small Letter A E

Part II

Base

Chapter 45

기본 골자

45.1 소개

Julia Base contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below. 함수들은 아래의 주제별로 묶여있다.

몇몇의 기본 사항은 다음과 같다:

- 모듈의 함수를 사용하기 위해서는 `import Module`로 모듈을 가지고 와서(import), `Module.fn(x)`의 형식으로 사용하면 된다.
- 다른 방법으로, `using Module`을 사용하면 현재의 namespace에서 `Module`의 모든 (all exported) 함수를 사용할 수 있다.
- 관습적으로(by convention), 느낌표(`!`)로 그 이름이 끝나는 함수는 전달받은 전달인자(arguments)의 값을 바꾼다. 몇몇 함수는 바꾸는 경우(e.g., `sort!`)와 바꾸지 않는 경우(`sort`)의 두 가지 버전을 모두 갖는다.

45.2 Getting Around

`Base.exit` – Function.

```
exit(code=0)
```

Stop the program with an exit code. The default exit code is zero, indicating that the program completed successfully. In an interactive session, `exit()` can be called with the keyboard shortcut `^D`.

source

`Base.atexit` – Function.

```
atexit(f)
```

Register a zero-argument function `f()` to be called at process exit. `atexit()` hooks are called in last in first out (LIFO) order and run before object finalizers.

Exit hooks are allowed to call `exit(n)`, in which case Julia will exit with exit code `n` (instead of the original exit code). If more than one exit hook calls `exit(n)`, then Julia will exit with the exit code corresponding to the last called exit hook that calls `exit(n)`. (Because exit hooks are called in LIFO order, "last called" is equivalent to "first registered".)

source

Base.isinteractive – Function.

```
isinteractive() -> Bool
```

Determine whether Julia is running an interactive session.

source

Base.summarysize – Function.

```
Base.summarysize(obj; exclude=Union{...}, chargeall=Union{...}) -> Int
```

Compute the amount of memory, in bytes, used by all unique objects reachable from the argument.

Keyword Arguments

- `exclude`: specifies the types of objects to exclude from the traversal.
- `chargeall`: specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

source

Base.require – Function.

```
require(into::Module, module::Symbol)
```

This function is part of the implementation of `using` / `import`, if a module is not already defined in `Main`. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the `Main` module, on every active node, searching standard locations for files. `require` is considered a top-level operation, so it sets the current `include` path but does not use it to search for files (see help for `include`). This function is typically used to load library code, and is implicitly called by `using` to load packages.

When searching for files, `require` first looks for package code in the global array `LOAD_PATH`. `require` is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

For more details regarding code loading, see the manual sections on modules and parallel computing.

source

`Base.compilecache` – Function.

```
Base.compilecache(module::PkgId)
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in `DEPOT_PATH[1]/compiled`. See Module initialization and precompilation for important notes.

source

`Base.__precompile__` – Function.

```
__precompile__(isprecompilable::Bool)
```

Specify whether the file calling this function is precompilable, defaulting to `true`. If a module or file is not safely precompilable, it should call `__precompile__(false)` in order to throw an error if Julia attempts to precompile it.

source

`Base.include` – Function.

```
Base.include([mapexpr::Function,] [m::Module,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of module `m`. Every module (except those defined with `baremodule`) has its own definition of `include` omitting the `m` argument, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

source

`Base.MainInclude.include` – Function.

```
include([mapexpr::Function,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of the containing module. Every module (except those defined with `baremodule`) has its own definition of `include`, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Use `Base.include` to evaluate a file into another module.

source

`Base.include_string` – Function.

```
include_string([mapexpr::Function,] m::Module, code::AbstractString, filename::AbstractString="string")
```

Like `include`, except reads code from the given string rather than from a file.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `code`, the `include_string` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

source

`Base.include_dependency` – Function.

```
include_dependency(path::AbstractString)
```

In a module, declare that the file specified by `path` (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via `include`. It has no effect outside of compilation.

source

`Base.which` – Method.

```
which(f, types)
```

Returns the method of `f` (a `Method` object) that would be called for arguments of the given `types`.

If `types` is an abstract type, then the method that would be called by `invoke` is returned.

source

`Base.methods` – Function.

```
methods(f, [types], [module])
```

Return the method table for `f`.

If `types` is specified, return an array of methods whose types match. If `module` is specified, return an array of methods defined in that module. A list of modules can also be specified as an array.

> Julia 1.4
>
> At least Julia 1.4 is required for specifying a module.

source

`Base.@show` – Macro.

```
@show
```

Show an expression and result, returning the result. See also `show`.

source

`ans` – Keyword.

```
ans
```

A variable referring to the last computed value, automatically set at the interactive prompt.

source

45.3 주요 단어들

아래의 단어 목록은 줄리아의 예약어(reserved keyword) 목록이다: `baremodule`, `begin`, `break`, `catch`, `const`, `continue`, `do`, `else`, `elseif`, `end`, `export`, `false`, `finally`, `for`, `function`, `global`, `if`, `import`, `let`, `local`, `macro`, `module`, `quote`, `return`, `struct`, `true`, `try`, `using`, `while`. 이 단어들은 변수 이름으로 사용하는 것이 불가하다.

아래의 연속된 두 단어짜리 구 전체는 예약어이다 (따라서 변수 이름으로 사용할 수 없다.): `abstract type`, `mutable struct`, `primitive type`. 하지만, 아래의 이름으로 변수를 만드는 것은 가능하다: `abstract`, `mutable`, `primitive`, and `type`.

마지막으로, Finally, `where`는 parametric method나 type 정의를 적을 때 중위 연산자(infix operator)로 구문분석된다(is parsed). `in`과 `isa` 또한 중위 연산자로 구문분석된다. 그러나 `where`, `in` 혹은 `isa`로 변수 이름을 짓는 것은 허용된다.

`module` – Keyword.

```
module
```

`module` declares a `Module`, which is a separate global variable workspace. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting). Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. See the manual section about modules for more details.

Examples

```
module Foo
import Base.show
export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1
show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

source

`export` – Keyword.

```
export
```

`export` is used within modules to tell Julia which functions should be made available to the user. For example: `export foo` makes the name `foo` available when `using` the module. See the manual section about modules for details.

source

`import` – Keyword.

```
import
```

`import Foo` will load the module or package `Foo`. Names from the imported `Foo` module can be accessed with dot syntax (e.g. `Foo.foo` to access the name `foo`). See the manual section about modules for details.

source

`using` – Keyword.

```
using
```

`using Foo` will load the module or package `Foo` and make its `export`ed names available for direct use. Names can also be used via dot syntax (e.g. `Foo.foo` to access the name `foo`), whether they are `export`ed or not. See the manual section about modules for details.

source

`baremodule` – Keyword.

```
baremodule
```

`baremodule` declares a module that does not contain `using Base` or a definition of `eval`. It does still import `Core`.

source

`function` – Keyword.

```
function
```

Functions are defined with the `function` keyword:

```
function add(a, b)
    return a + b
end
```

Or the short form notation:

```
add(a, b) = a + b
```

The use of the return keyword is exactly the same as in other languages, but is often optional. A function without an explicit **return** statement will return the last expression in the function body.

source

macro – Keyword.

```
macro
```

macro defines a method for inserting generated code into a program. A macro maps a sequence of argument expressions to a returned expression, and the resulting expression is substituted directly into the program at the point where the macro is invoked. Macros are a way to run generated code without calling eval, since the generated code instead simply becomes part of the surrounding program. Macro arguments may include expressions, literal values, and symbols. Macros can be defined for variable number of arguments (varargs), but do not accept keyword arguments.

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!

julia> macro saylots(x...)
           return :( println("Say: ", $(x...)) )
       end
@saylots (macro with 1 method)

julia> @saylots "hey " "there " "friend"
Say: hey there friend
```

source

return – Keyword.

```
return
```

`return x` causes the enclosing function to exit early, passing the given value `x` back to its caller. `return` by itself with no value is equivalent to `return nothing` (see nothing).

```
function compare(a, b)
    a == b && return "equal to"
    a < b ? "less than" : "greater than"
end
```

In general you can place a `return` statement anywhere within a function body, including within deeply nested loops or conditionals, but be careful with `do` blocks. For example:

```
function test1(xs)
    for x in xs
        iseven(x) && return 2x
    end
end

function test2(xs)
    map(xs) do x
        iseven(x) && return 2x
        x
    end
end
```

In the first example, the return breaks out of `test1` as soon as it hits an even number, so `test1([5,6,7])` returns `12`.

You might expect the second example to behave the same way, but in fact the `return` there only breaks out of the inner function (inside the `do` block) and gives a value back to `map`. `test2([5,6,7])` then returns `[5,12,7]`.

When used in a top-level expression (i.e. outside any function), `return` causes the entire current top-level expression to terminate early.

source

do – Keyword.

```
do
```

Create an anonymous function and pass it as the first argument to a function call. For example:

```
map(1:10) do x
    2x
end
```

is equivalent to `map(x->2x, 1:10)`.

Use multiple arguments like so:

```
map(1:10, 11:20) do x, y
    x + y
end
```

source

`begin` – Keyword.

```
begin
```

`begin...end` denotes a block of code.

```
begin
    println("Hello, ")
    println("World!")
end
```

Usually `begin` will not be necessary, since keywords such as `function` and `let` implicitly begin blocks of code. See also `;`.

source

`end` – Keyword.

```
end
```

`end` marks the conclusion of a block of expressions, for example `module`, `struct`, `mutable struct`, `begin`, `let`, `for` etc. `end` may also be used when indexing into an array to represent the last index of a dimension.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
```

```
 3  4

julia> A[end, :]
2-element Array{Int64,1}:
 3
 4
```

source

let – Keyword.

```
let
```

let statements allocate new variable bindings each time they run. Whereas an assignment modifies an existing value location, **let** creates new locations. This difference is only detectable in the case of variables that outlive their scope via closures. The **let** syntax accepts a comma-separated series of assignments and variable names:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x`, since the two `x` variables are distinct and have separate storage.

source

if – Keyword.

```
if/elseif/else
```

if/elseif/else performs conditional evaluation, which allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the **if/elseif/else** conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression is true, the **`else`** block is evaluated. The **`elseif`** and **`else`** blocks are optional, and as many **`elseif`** blocks as desired can be used.

source

`for` – Keyword.

```
for
```

`for` loops repeatedly evaluate a block of statements while iterating over a sequence of values.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

source

`while` – Keyword.

```
while
```

`while` loops repeatedly evaluate a conditional expression, and continue evaluating the body of the while loop as long as the expression remains true. If the condition expression is false when the while loop is first reached, the body is never evaluated.

Examples

```
julia> i = 1
1

julia> while i < 5
           println(i)
           global i += 1
       end
1
```

```
2
3
4
```

source

`break` – Keyword.

```
break
```

Break out of a loop immediately.

Examples

```
julia> i = 0
0

julia> while true
           global i += 1
           i > 5 && break
           println(i)
       end
1
2
3
4
5
```

source

`continue` – Keyword.

```
continue
```

Skip the rest of the current loop iteration.

Examples

```
julia> for i = 1:6
           iseven(i) && continue
           println(i)
       end
```

```
1
3
5
```

source

`try` – Keyword.

```
try/catch
```

A `try/catch` statement allows intercepting errors (exceptions) thrown by `throw` so that program execution can continue. For example, the following code attempts to write a file, but warns the user and proceeds instead of terminating execution if the file cannot be written:

```
try
    open("/danger", "w") do f
        println(f, "Hello")
    end
catch
    @warn "Could not write file."
end
```

or, when the file cannot be read into a variable:

```
lines = try
    open("/danger", "r") do f
        readlines(f)
    end
catch
    @warn "File not found."
end
```

The syntax `catch e` (where `e` is any variable) assigns the thrown exception object to the given variable within the `catch` block.

The power of the `try/catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions.

source

`finally` – Keyword.

```
finally
```

Run some code when a given block of code exits, regardless of how it exits. For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    operate_on_file(f)
finally
    close(f)
end
```

When control leaves the `try` block (for example, due to a `return`, or just finishing normally), `close(f)` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `catch` block may be combined with `try` and `finally` as well. In this case the `finally` block will run after `catch` has handled the error.

source

`quote` – Keyword.

```
quote
```

`quote` creates multiple expression objects in a block without using the explicit `Expr` constructor. For example:

```
ex = quote
    x = 1
    y = 2
    x + y
end
```

Unlike the other means of quoting, `:( ... )`, this form introduces `QuoteNode` elements to the expression tree, which must be considered when directly manipulating the tree. For other purposes, `:( ... )` and `quote .. end` blocks are treated identically.

source

`local` – Keyword.

```
local
```

`local` introduces a new local variable. See the manual section on variable scoping for more information.

Examples

```
julia> function foo(n)
           x = 0
           for i = 1:n
               local x # introduce a loop-local x
               x = i
           end
           x
       end
foo (generic function with 1 method)

julia> foo(10)
0
```

source

`global` – Keyword.

```
global
```

`global x` makes `x` in the current scope and its inner scopes refer to the global variable of that name. See the manual section on variable scoping for more information.

Examples

```
julia> z = 3
3

julia> function foo()
           global z = 6 # use the z variable defined outside foo
       end
foo (generic function with 1 method)

julia> foo()
6

julia> z
6
```

source

const – Keyword.

```
const
```

`const` is used to declare global variables whose values will not change. In almost all code (and particularly performance sensitive code) global variables should be declared constant in this way.

```
const x = 5
```

Multiple variables can be declared within a single `const`:

```
const y, z = 7, 11
```

Note that `const` only applies to one `=` operation, therefore `const x = y = 1` declares `x` to be constant but not `y`. On the other hand, `const x = const y = 1` declares both `x` and `y` constant.

Note that "constant-ness" does not extend into mutable containers; only the association between a variable and its value is constant. If `x` is an array or dictionary (for example) you can still modify, add, or remove elements.

In some cases changing the value of a `const` variable gives a warning instead of an error. However, this can produce unpredictable behavior or corrupt the state of your program, and so should be avoided. This feature is intended only for convenience during interactive use.

source

struct – Keyword.

```
struct
```

The most commonly used kind of type in Julia is a struct, specified as a name and a set of fields.

```
struct Point
    x
    y
end
```

Fields can have type restrictions, which may be parameterized:

```
struct Point{X}
    x::X
    y::Float64
end
```

A struct can also declare an abstract super type via `<:` syntax:

```
struct Point <: AbstractPoint
    x
    y
end
```

`struct`s are immutable by default; an instance of one of these types cannot be modified after construction. Use `mutable struct` instead to declare a type whose instances can be modified.

See the manual section on Composite Types for more details, such as how to define constructors.

source

`mutable struct` – Keyword.

```
mutable struct
```

`mutable struct` is similar to `struct`, but additionally allows the fields of the type to be set after construction. See the manual section on Composite Types for more information.

source

`abstract type` – Keyword.

```
abstract type
```

`abstract type` declares a type that cannot be instantiated, and serves only as a node in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. Abstract types form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations. For example:

```
abstract type Number end
abstract type Real <: Number end
```

`Number` has no supertype, whereas `Real` is an abstract subtype of `Number`.

source

`primitive type` – Keyword.

```
primitive type
```

primitive type declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some example built-in primitive type declarations:

```
primitive type Char 32 end
primitive type Bool <: Integer 8 end
```

The number after the name indicates how many bits of storage the type requires. Currently, only sizes that are multiples of 8 bits are supported. The `Bool` declaration shows how a primitive type can be optionally declared to be a subtype of some supertype.

source

`where` – Keyword.

```
where
```

The **where** keyword creates a type that is an iterated union of other types, over all values of some variable. For example **Vector{T} where T<:Real** includes all `Vector`s where the element type is some kind of **Real** number.

The variable bound defaults to `Any` if it is omitted:

```
Vector{T} where T    # short for `where T<:Any`
```

Variables can also have lower bounds:

```
Vector{T} where T>:Int
Vector{T} where Int<:T<:Real
```

There is also a concise syntax for nested **where** expressions. For example, this:

```
Pair{T, S} where S<:Array{T} where T<:Number
```

can be shortened to:

```
Pair{T, S} where {T<:Number, S<:Array{T}}
```

This form is often found on method signatures.

Note that in this form, the variables are listed outermost-first. This matches the order in which variables are substituted when a type is "applied" to parameter values using the syntax **T{p1, p2, ...}**.

source

... – Keyword.

```
...
```

The "splat" operator, ..., represents a sequence of arguments. ... can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. ... can also be used to apply a function to a sequence of arguments.

Examples

```
julia> add(xs...) = reduce(+, xs)
add (generic function with 1 method)

julia> add(1, 2, 3, 4, 5)
15

julia> add([1, 2, 3]...)
6

julia> add(7, 1:100..., 1000:1100...)
111107
```

source

; – Keyword.

```
;
```

; has a similar role in Julia as in many C-like languages, and is used to delimit the end of the previous statement. ; is not necessary after new lines, but can be used to separate statements on a single line or to join statements into a single expression. ; is also used to suppress output printing in the REPL and similar interfaces.

Examples

```
julia> function foo()
           x = "Hello, "; x *= "World!"
           return x
       end
foo (generic function with 1 method)

julia> bar() = (x = "Hello, Mars!"; return x)
```

```
bar (generic function with 1 method)

julia> foo();

julia> bar()
"Hello, Mars!"
```

source

= – Keyword.

```
=
```

= is the assignment operator.

- For variable a and expression b, a = b makes a refer to the value of b.
- For functions f(x), f(x) = x defines a new function constant f, or adds a new method to f if f is already defined; this usage is equivalent to function f(x); x; end.
- a[i] = v calls setindex!(a,v,i).
- a.b = c calls setproperty!(a,:b,c).
- Inside a function call, f(a=b) passes b as the value of keyword argument a.
- Inside parentheses with commas, (a=1,) constructs a NamedTuple.

Examples

Assigning a to b does not create a copy of b; instead use copy or deepcopy.

```
julia> b = [1]; a = b; b[1] = 2; a
1-element Array{Int64,1}:
 2

julia> b = [1]; a = copy(b); b[1] = 2; a
1-element Array{Int64,1}:
 1
```

Collections passed to functions are also not copied. Functions can modify (mutate) the contents of the objects their arguments refer to. (The names of functions which do this are conventionally suffixed with '!'.)

```
julia> function f!(x); x[:] .+= 1; end
f! (generic function with 1 method)

julia> a = [1]; f!(a); a
1-element Array{Int64,1}:
 2
```

Assignment can operate on multiple variables in parallel, taking values from an iterable:

```
julia> a, b = 4, 5
(4, 5)

julia> a, b = 1:3
1:3

julia> a, b
(1, 2)
```

Assignment can operate on multiple variables in series, and will return the value of the right-hand-most expression:

```
julia> a = [1]; b = [2]; c = [3]; a = b = c
1-element Array{Int64,1}:
 3

julia> b[1] = 2; a, b, c
([2], [2], [2])
```

Assignment at out-of-bounds indices does not grow a collection. If the collection is a `Vector` it can instead be grown with `push!` or `append!`.

```
julia> a = [1, 1]; a[3] = 2
ERROR: BoundsError: attempt to access 2-element Array{Int64,1} at index [3]
[...]

julia> push!(a, 2, 3)
4-element Array{Int64,1}:
 1
 1
 2
 3
```

Assigning `[]` does not eliminate elements from a collection; instead use `filter!`.

```
julia> a = collect(1:3); a[a .<= 1] = []
ERROR: DimensionMismatch("tried to assign 0 elements to 1 destinations")
[...]

julia> filter!(x -> x > 1, a) # in-place & thus more efficient than a = a[a .> 1]
2-element Array{Int64,1}:
 2
 3
```

source

45.4 Standard Modules

`Main` – Module.

```
Main
```

`Main` is the top-level module, and Julia starts with `Main` set as the current module. Variables defined at the prompt go in `Main`, and `varinfo` lists variables in `Main`.

```
julia> @__MODULE__
Main
```

source

`Core` – Module.

```
Core
```

`Core` is the module that contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies `using Core`, since you can't do anything without those definitions.

source

`Base` – Module.

```
Base
```

The base library of Julia. `Base` is a module that contains basic functionality (the contents of `base/`). All modules implicitly contain `using Base`, since this is needed in the vast majority of cases.

source

45.5 Base Submodules

`Base.Broadcast` – Module.

```
Base.Broadcast
```

Module containing the broadcasting implementation.

source

`Base.Docs` – Module.

```
Docs
```

The `Docs` module provides the `@doc` macro which can be used to set and retrieve documentation metadata for Julia objects.

Please see the manual section on documentation for more information.

source

`Base.Iterators` – Module.

Methods for working with Iterators.

source

`Base.Libc` – Module.

Interface to libc, the C standard library.

source

`Base.Meta` – Module.

Convenience functions for metaprogramming.

source

`Base.StackTraces` – Module.

Tools for collecting and manipulating stack traces. Mainly used for building errors.

source

Base.Sys – Module.

Provide methods for retrieving information about hardware and the operating system.

source

Base.Threads – Module.

Multithreading support.

source

Base.GC – Module.

```
Base.GC
```

Module with garbage collection utilities.

source

45.6 All Objects

Core.:=== – Function.

```
===(x,y) -> Bool
≡(x,y) -> Bool
```

Determine whether x and y are identical, in the sense that no program could distinguish them. First the types of x and y are compared. If those are identical, mutable objects are compared by address in memory and immutable objects (such as numbers) are compared by contents at the bit level. This function is sometimes called "egal". It always returns a `Bool` value.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a == b
true
```

```
julia> a === b
false

julia> a === a
true
```

source

Core.isa – Function.

```
isa(x, type) -> Bool
```

Determine whether `x` is of the given `type`. Can also be used as an infix operator, e.g. `x isa type`.

Examples

```
julia> isa(1, Int)
true

julia> isa(1, Matrix)
false

julia> isa(1, Char)
false

julia> isa(1, Number)
true

julia> 1 isa Number
true
```

source

Base.isequal – Function.

```
isequal(x, y)
```

Similar to ==, except for the treatment of floating point numbers and of missing values. `isequal` treats all floating-point `NaN` values as equal to each other, treats `-0.0` as unequal to `0.0`, and missing as equal to `missing`. Always returns a `Bool` value.

Implementation

The default implementation of `isequal` calls `==`, so a type that does not involve floating-point values generally only needs to define `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x,y)` must imply that `hash(x) == hash(y)`.

This typically means that types for which a custom `==` or `isequal` method exists must implement a corresponding `hash` method (and vice versa). Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Scalar types generally do not need to implement `isequal` separate from `==`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

Examples

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
true

julia> isequal(0.0, -0.0)
false
```

source

```
isequal(x)
```

Create a function that compares its argument to `x` using `isequal`, i.e. a function equivalent to `y -> isequal(y, x)`.

The returned function is of type `Base.Fix2{typeof(isequal)}`, which can be used to implement specialized methods.

source

`Base.isless` – Function.

```
isless(x, y)
```

Test whether `x` is less than `y`, according to a fixed total order. `isless` is not defined on all pairs of values `(x, y)`. However, if it is defined, it is expected to satisfy the following:

- If `isless(x, y)` is defined, then so is `isless(y, x)` and `isequal(x, y)`, and exactly one of those three yields `true`.
- The relation defined by `isless` is transitive, i.e., `isless(x, y) && isless(y, z)` implies `isless(x, z)`.

Values that are normally unordered, such as `NaN`, are ordered in an arbitrary but consistent fashion. `missing` values are ordered last.

This is the default comparison used by `sort`.

Implementation

Non-numeric types with a total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`. Types with a partial order should implement `<`.

Examples

“‘jldoctest julia> isless(1, 3) true

julia> isless("Red", "Blue") false “‘

source

`Core.ifelse` – Function.

```
ifelse(condition::Bool, x, y)
```

Return `x` if `condition` is `true`, otherwise return `y`. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

```
julia> ifelse(1 > 2, 1, 2)
2
```

source

`Core.typeassert` – Function.

```
typeassert(x, type)
```

Throw a TypeError unless x isa type. The syntax x::type calls this function.

Examples

```
julia> typeassert(2.5, Int)
ERROR: TypeError: in typeassert, expected Int64, got a value of type Float64
Stacktrace:
[...]
```

source

Core.typeof – Function.

```
typeof(x)
```

Get the concrete type of x.

Examples

```
julia> a = 1//2;

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Array{Float64,2}
```

source

Core.tuple – Function.

```
tuple(xs...)
```

Construct a tuple of the given objects.

Examples

```
julia> tuple(1, 'a', pi)
(1, 'a', π)
```

source

Base.ntuple — Function.

```
ntuple(f::Function, n::Integer)
```

Create a tuple of length n, computing each element as f(i), where i is the index of the element.

Examples

```
julia> ntuple(i -> 2*i, 4)
(2, 4, 6, 8)
```

source

Base.objectid — Function.

```
objectid(x)
```

Get a hash value for x based on object identity. objectid(x)==objectid(y) if x === y.

source

Base.hash — Function.

```
hash(x[, h::UInt])
```

Compute an integer hash code such that isequal(x,y) implies hash(x)==hash(y). The optional second argument h is a hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument hash method recursively in order to mix hashes of the contents with each other (and with h). Typically, any type that implements hash should also implement its own == (hence isequal) to guarantee the property mentioned above. Types supporting subtraction (operator -) should also implement widen, which is required to hash values inside heterogeneous arrays.

source

Base.finalizer — Function.

```
finalizer(f, x)
```

Register a function f(x) to be called when there are no program-accessible references to x, and return x. The type of x must be a mutable struct, otherwise the behavior of this function is unpredictable.

`f` must not cause a task switch, which excludes most I/O operations such as `println`. Using the `@async` macro (to defer context switching to outside of the finalizer) or `ccall` to directly invoke IO functions in C may be helpful for debugging purposes.

Examples

```
finalizer(my_mutable_struct) do x
    @async println("Finalizing $x.")
end

finalizer(my_mutable_struct) do x
    ccall(:jl_safe_printf, Cvoid, (Cstring, Cstring), "Finalizing %s.", repr(x))
end
```

source

`Base.finalize` — Function.

```
finalize(x)
```

Immediately run finalizers registered for object `x`.

source

`Base.copy` — Function.

```
copy(x)
```

Create a shallow copy of `x`: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

source

`Base.deepcopy` — Function.

```
deepcopy(x)
```

Create a deep copy of `x`: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling `deepcopy` on an object should generally have the same effect as serializing and then deserializing it.

While it isn't normally necessary, user-defined types can override the default `deepcopy` behavior by defining a specialized version of the function `deepcopy_internal(x::T, dict::IdDict)` (which shouldn't otherwise be used),

where `T` is the type to be specialized for, and `dict` keeps track of objects copied so far within the recursion. Within the definition, `deepcopy_internal` should be used in place of `deepcopy`, and the `dict` variable should be updated as appropriate before returning.

source

`Base.getproperty` – Function.

```
getproperty(value, name::Symbol)
```

The syntax `a.b` calls `getproperty(a, :b)`.

Examples

```
julia> struct MyType
           x
       end

julia> function Base.getproperty(obj::MyType, sym::Symbol)
           if sym === :special
               return obj.x + 1
           else # fallback to getfield
               return getfield(obj, sym)
           end
       end

julia> obj = MyType(1);

julia> obj.special
2

julia> obj.x
1
```

See also `propertynames` and `setproperty!`.

source

`Base.setproperty!` – Function.

```
setproperty!(value, name::Symbol, x)
```

The syntax `a.b = c` calls `setproperty!(a, :b, c)`.

See also `propertynames` and `getproperty`.

source

`Base.propertynames` – Function.

```
propertynames(x, private=false)
```

Get a tuple or a vector of the properties (`x.property`) of an object `x`. This is typically the same as `fieldnames(typeof(x))`, but types that overload `getproperty` should generally overload `propertynames` as well to get the properties of an instance of the type.

`propertynames(x)` may return only "public" property names that are part of the documented interface of `x`. If you want it to also return "private" fieldnames intended for internal use, pass `true` for the optional second argument. REPL tab completion on `x.` shows only the `private=false` properties.

source

`Base.hasproperty` – Function.

```
hasproperty(x, s::Symbol)
```

Return a boolean indicating whether the object `x` has `s` as one of its own properties.

> Julia 1.2
>
> This function requires at least Julia 1.2.

source

`Core.getfield` – Function.

```
getfield(value, name::Symbol)
getfield(value, i::Int)
```

Extract a field from a composite `value` by name or position. See also `getproperty` and `fieldnames`.

Examples

```
julia> a = 1//2
1//2
```

```
julia> getfield(a, :num)
1

julia> a.num
1

julia> getfield(a, 1)
1
```

source

Core.setfield! – Function.

```
setfield!(value, name::Symbol, x)
```

Assign `x` to a named field in `value` of composite type. The `value` must be mutable and `x` must be a subtype of `fieldtype(typeof(value), name)`. See also setproperty!.

Examples

```
julia> mutable struct MyMutableStruct
           field::Int
       end

julia> a = MyMutableStruct(1);

julia> setfield!(a, :field, 2);

julia> getfield(a, :field)
2

julia> a = 1//2
1//2

julia> setfield!(a, :num, 3);
ERROR: setfield! immutable struct of type Rational cannot be changed
```

source

Core.isdefined – Function.

```
isdefined(m::Module, s::Symbol)
isdefined(object, s::Symbol)
isdefined(object, index::Int)
```

Tests whether a global variable or object field is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index.

To test whether an array element is defined, use `isassigned` instead.

See also `@isdefined`.

Examples

```
julia> isdefined(Base, :sum)
true

julia> isdefined(Base, :NonExistentMethod)
false

julia> a = 1//2;

julia> isdefined(a, 2)
true

julia> isdefined(a, 3)
false

julia> isdefined(a, :num)
true

julia> isdefined(a, :numerator)
false
```

source

`Base.@isdefined` – Macro.

```
@isdefined s -> Bool
```

Tests whether variable `s` is defined in the current scope.

See also `isdefined`.

Examples

```
julia> function f()
           println(@isdefined x)
           x = 3
           println(@isdefined x)
       end
f (generic function with 1 method)

julia> f()
false
true
```

source

Base.convert – Function.

```
convert(T, x)
```

Convert x to a value of type T.

If T is an Integer type, an InexactError will be raised if x is not representable by T, for example if x is not integer-valued, or is outside the range supported by T.

Examples

```
julia> convert(Int, 3.0)
3

julia> convert(Int, 3.5)
ERROR: InexactError: Int64(3.5)
Stacktrace:
[...]
```

If T is a AbstractFloat or Rational type, then it will return the closest value to x representable by T.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0
```

```
julia> convert(Rational{Int32}, x)
1//3

julia> convert(Rational{Int64}, x)
6004799503160661//18014398509481984
```

If `T` is a collection type and `x` a collection, the result of `convert(T, x)` may alias all or part of `x`.

```
julia> x = Int[1, 2, 3];

julia> y = convert(Vector{Int}, x);

julia> y === x
true
```

source

`Base.promote` – Function.

```
promote(xs...)
```

Convert all arguments to a common type, and return them all (as a tuple). If no arguments can be converted, an error is raised.

Examples

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)
```

source

`Base.oftype` – Function.

```
oftype(x, y)
```

Convert `y` to the type of `x` (`convert(typeof(x), y)`).

Examples

```
julia> x = 4;
```

```
julia> y = 3.;

julia> oftype(x, y)
3

julia> oftype(y, x)
4.0
```

source

Base.widen – Function.

```
widen(x)
```

If x is a type, return a "larger" type, defined so that arithmetic operations + and - are guaranteed not to overflow nor lose precision for any combination of values that type x can hold.

For fixed-size integer types less than 128 bits, widen will return a type with twice the number of bits.

If x is a value, it is converted to widen(typeof(x)).

Examples

```
julia> widen(Int32)
Int64

julia> widen(1.5f0)
1.5
```

source

Base.identity – Function.

```
identity(x)
```

The identity function. Returns its argument.

Examples

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

source

45.7 Properties of Types

Type relations

`Base.supertype` – Function.

```
supertype(T::DataType)
```

Return the supertype of DataType `T`.

Examples

```
julia> supertype(Int32)
Signed
```

source

`Core.:<:` – Function.

```
<:(T1, T2)
```

Subtype operator: returns `true` if and only if all values of type `T1` are also of type `T2`.

Examples

```
julia> Float64 <: AbstractFloat
true

julia> Vector{Int} <: AbstractArray
true

julia> Matrix{Float64} <: Matrix{AbstractFloat}
false
```

source

`Base.:>:` – Function.

```
>:(T1, T2)
```

Supertype operator, equivalent to `T2 <: T1`.

source

Base.typejoin – Function.

```
typejoin(T, S)
```

Return the closest common ancestor of T and S, i.e. the narrowest type from which they both inherit.

source

Base.typeintersect – Function.

```
typeintersect(T, S)
```

Compute a type that contains the intersection of T and S. Usually this will be the smallest such type or one close to it.

source

Base.promote_type – Function.

```
promote_type(type1, type2)
```

Promotion refers to converting values of mixed types to a single common type. promote_type represents the default promotion behavior in Julia when operators (usually mathematical) are given arguments of differing types. promote_type generally tries to return a type which can at least approximate most values of either input type without excessively widening. Some loss is tolerated; for example, promote_type(Int64, Float64) returns Float64 even though strictly, not all Int64 values can be represented exactly as Float64 values.

```
julia> promote_type(Int64, Float64)
Float64

julia> promote_type(Int32, Int64)
Int64

julia> promote_type(Float32, BigInt)
BigFloat

julia> promote_type(Int16, Float16)
Float16

julia> promote_type(Int64, Float16)
Float16
```

```
julia> promote_type(Int8, UInt16)
UInt16
```

source

Base.promote_rule – Function.

```
promote_rule(type1, type2)
```

Specifies what type should be used by promote when given values of types type1 and type2. This function should not be called directly, but should have definitions added to it for new types as appropriate.

source

Base.isdispatchtuple – Function.

```
isdispatchtuple(T)
```

Determine whether type T is a tuple "leaf type", meaning it could appear as a type signature in dispatch and has no subtypes (or supertypes) which could appear in a call.

source

Declared structure

Base.isimmutable – Function.

```
isimmutable(v) -> Bool
```

> Warning
>
> Consider using !ismutable(v) instead, as isimmutable(v) will be replaced by !ismutable(v) in a future release. (Since Julia 1.5)

Return true iff value v is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of DataType is mutable.

Examples

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

source

Base.isabstracttype – Function.

```
isabstracttype(T)
```

Determine whether type T was declared as an abstract type (i.e. using the abstract keyword).

Examples

```
julia> isabstracttype(AbstractArray)
true

julia> isabstracttype(Vector)
false
```

source

Base.isprimitivetype – Function.

```
isprimitivetype(T) -> Bool
```

Determine whether type T was declared as a primitive type (i.e. using the primitive keyword).

source

Base.issingletontype – Function.

```
Base.issingletontype(T)
```

Determine whether type T has exactly one possible instance; for example, a struct type with no fields.

source

Base.isstructtype – Function.

```
isstructtype(T) -> Bool
```

Determine whether type T was declared as a struct type (i.e. using the struct or mutable struct keyword).

source

Base.nameof – Method.

```
nameof(t::DataType) -> Symbol
```

Get the name of a (potentially `UnionAll`-wrapped `DataType` (without its parent module) as a symbol.

Examples

```
julia> module Foo
           struct S{T}
           end
       end
Foo

julia> nameof(Foo.S{T} where T)
:S
```

source

Base.fieldnames – Function.

```
fieldnames(x::DataType)
```

Get a tuple with the names of the fields of a `DataType`.

Examples

```
julia> fieldnames(Rational)
(:num, :den)
```

source

Base.fieldname – Function.

```
fieldname(x::DataType, i::Integer)
```

Get the name of field `i` of a `DataType`.

Examples

```
julia> fieldname(Rational, 1)
:num

julia> fieldname(Rational, 2)
:den
```

source

Base.hasfield – Function.

```
hasfield(T::Type, name::Symbol)
```

Return a boolean indicating whether T has name as one of its own fields.

Julia 1.2

This function requires at least Julia 1.2.

source

Memory layout

Base.sizeof – Method.

```
sizeof(T::DataType)
sizeof(obj)
```

Size, in bytes, of the canonical binary representation of the given DataType T, if any. Size, in bytes, of object obj if it is not DataType.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(ComplexF64)
16

julia> sizeof(1.0)
8

julia> sizeof([1.0:10.0;])
80
```

If DataType T does not have a specific size, an error is thrown.

```
julia> sizeof(AbstractArray)
ERROR: Abstract type AbstractArray does not have a definite size.
Stacktrace:
[...]
```

source

Base.isconcretetype – Function.

```
isconcretetype(T)
```

Determine whether type `T` is a concrete type, meaning it could have direct instances (values `x` such that `typeof(x) === T`).

Examples

```
julia> isconcretetype(Complex)
false

julia> isconcretetype(Complex{Float32})
true

julia> isconcretetype(Vector{Complex})
true

julia> isconcretetype(Vector{Complex{Float32}})
true

julia> isconcretetype(Union{})
false

julia> isconcretetype(Union{Int,String})
false
```

source

Base.isbits – Function.

```
isbits(x)
```

Return `true` if `x` is an instance of an `isbitstype` type.

source

Base.isbitstype – Function.

```
isbitstype(T)
```

Return `true` if type `T` is a "plain data" type, meaning it is immutable and contains no references to other values, only `primitive` types and other `isbitstype` types. Typical examples are numeric types such as `UInt8`, `Float64`, and `Complex{Float64}`. This category of types is significant since they are valid as type parameters, may not track `isdefined` / `isassigned` status, and have a defined layout that is compatible with C.

Examples

```
julia> isbitstype(Complex{Float64})
true

julia> isbitstype(Complex)
false
```

source

`Core.fieldtype` – Function.

```
fieldtype(T, name::Symbol | index::Int)
```

Determine the declared type of a field (specified by name or index) in a composite DataType `T`.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

source

`Base.fieldtypes` – Function.

```
fieldtypes(T::Type)
```

The declared types of all fields in a composite DataType `T` as a tuple.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtypes(Foo)
(Int64, String)
```

source

Base.fieldcount – Function.

```
fieldcount(t::Type)
```

Get the number of fields that an instance of the given type would have. An error is thrown if the type is too abstract to determine this.

source

Base.fieldoffset – Function.

```
fieldoffset(type, i)
```

The byte offset of field `i` of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i), fieldtype(T,i)) for i = 1:fieldcount(T)];

julia> structinfo(Base.Filesystem.StatStruct)
12-element Array{Tuple{UInt64,Symbol,DataType},1}:
 (0x0000000000000000, :device, UInt64)
 (0x0000000000000008, :inode, UInt64)
 (0x0000000000000010, :mode, UInt64)
 (0x0000000000000018, :nlink, Int64)
 (0x0000000000000020, :uid, UInt64)
 (0x0000000000000028, :gid, UInt64)
```

```
(0x0000000000000030, :rdev, UInt64)
(0x0000000000000038, :size, Int64)
(0x0000000000000040, :blksize, Int64)
(0x0000000000000048, :blocks, Int64)
(0x0000000000000050, :mtime, Float64)
(0x0000000000000058, :ctime, Float64)
```

source

Base.datatype_alignment – Function.

```
Base.datatype_alignment(dt::DataType) -> Int
```

Memory allocation minimum alignment for instances of this type. Can be called on any `isconcretetype`.

source

Base.datatype_haspadding – Function.

```
Base.datatype_haspadding(dt::DataType) -> Bool
```

Return whether the fields of instances of this type are packed in memory, with no intervening padding bytes. Can be called on any `isconcretetype`.

source

Base.datatype_pointerfree – Function.

```
Base.datatype_pointerfree(dt::DataType) -> Bool
```

Return whether instances of this type can contain references to gc-managed memory. Can be called on any `isconcretetype`.

source

Special values

Base.typemin – Function.

```
typemin(T)
```

The lowest value representable by the given (real) numeric DataType `T`.

Examples

```
julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
-Inf32
```

source

Base.typemax – Function.

```
typemax(T)
```

The highest value representable by the given (real) numeric `DataType`.

Examples

```
julia> typemax(Int8)
127

julia> typemax(UInt32)
0xffffffff
```

source

Base.floatmin – Function.

```
floatmin(T = Float64)
```

Return the smallest positive normal number representable by the floating-point type `T`.

Examples

```
julia> floatmin(Float16)
Float16(6.104e-5)

julia> floatmin(Float32)
1.1754944f-38

julia> floatmin()
2.2250738585072014e-308
```

source

Base.floatmax – Function.

```
floatmax(T = Float64)
```

Return the largest finite number representable by the floating-point type T.

Examples

```
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38

julia> floatmax()
1.7976931348623157e308
```

source

Base.maxintfloat – Function.

```
maxintfloat(T=Float64)
```

The largest consecutive integer-valued floating-point number that is exactly represented in the given floating-point type T (which defaults to Float64).

That is, maxintfloat returns the smallest positive integer-valued floating-point number n such that n+1 is not exactly representable in the type T.

When an Integer-type value is needed, use Integer(maxintfloat(T)).

source

```
maxintfloat(T, S)
```

The largest consecutive integer representable in the given floating-point type T that also does not exceed the maximum integer representable by the integer type S. Equivalently, it is the minimum of maxintfloat(T) and typemax(S).

source

Base.eps – Method.

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the machine epsilon of the floating point type `T` (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the relative error of `T`, it is a "dimensionless" quantity like one.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

source

`Base.eps` – Method.

```
eps(x::AbstractFloat)
```

Return the unit in last place (ulp) of `x`. This is the distance between consecutive representable floating point values at `x`. In most cases, if the distance on either side of `x` is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \mathsf{eps}(x)/2.$$

Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

source

Base.instances – Function.

```
instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `@enum`).

Example

```
julia> @enum Color red blue green

julia> instances(Color)
(red, blue, green)
```

source

45.8 Special Types

Core.Any – Type.

```
Any::DataType
```

`Any` is the union of all types. It has the defining property `isa(x, Any) == true` for any `x`. `Any` therefore describes the entire universe of possible values. For example `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other integer types.

source

`Core.Union` – Type.

```
Union{Types...}
```

A type union is an abstract type which includes all instances of any of its argument types. The empty union `Union{}` is the bottom type of Julia.

Examples

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got a value of type Float64
```

source

`Union{}` – Keyword.

```
Union{}
```

`Union{}`, the empty `Union` of types, is the type that has no values. That is, it has the defining property `isa(x, Union{}) == false` for any `x`. `Base.Bottom` is defined as its alias and the type of `Union{}` is `Core.TypeofBottom`.

Examples

```
julia> isa(nothing, Union{})
false
```

source

Core.UnionAll — Type.

```
UnionAll
```

A union of types over all values of a type parameter. `UnionAll` is used to describe parametric types where the values of some parameters are not known.

Examples

```
julia> typeof(Vector)
UnionAll

julia> typeof(Vector{Int})
DataType
```

source

Core.Tuple — Type.

```
Tuple{Types...}
```

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters.

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are. Tuples do not have field names; fields are only accessed by index.

See the manual section on Tuple Types.

source

Core.NamedTuple — Type.

```
NamedTuple
```

`NamedTuple`s are, as their name suggests, named `Tuple`s. That is, they're a tuple-like collection of values, where each entry has a unique name, represented as a `Symbol`. Like `Tuple`s, `NamedTuple`s are immutable; neither the names nor the values can be modified in place after construction.

Accessing the value associated with a name in a named tuple can be done using field access syntax, e.g. `x.a`, or using `getindex`, e.g. `x[:a]`. A tuple of the names can be obtained using `keys`, and a tuple of the values can be obtained using `values`.

Note

Iteration over `NamedTuple`s produces the values without the names. (See example below.) To iterate over the name-value pairs, use the `pairs` function.

The `@NamedTuple` macro can be used for conveniently declaring `NamedTuple` types.

Examples

```
julia> x = (a=1, b=2)
(a = 1, b = 2)

julia> x.a
1

julia> x[:a]
1

julia> keys(x)
(:a, :b)

julia> values(x)
(1, 2)

julia> collect(x)
2-element Array{Int64,1}:
 1
 2

julia> collect(pairs(x))
2-element Array{Pair{Symbol,Int64},1}:
 :a => 1
 :b => 2
```

In a similar fashion as to how one can define keyword arguments programmatically, a named tuple can be created by giving a pair `name::Symbol => value` or splatting an iterator yielding such pairs after a semicolon inside a tuple literal:

```
julia> (; :a => 1)
(a = 1,)
```

```
julia> keys = (:a, :b, :c); values = (1, 2, 3);

julia> (; zip(keys, values)...)
(a = 1, b = 2, c = 3)
```

As in keyword arguments, identifiers and dot expressions imply names:

```
julia> x = 0
0

julia> t = (; x)
(x = 0,)

julia> (; t.x)
(x = 0,)
```

Julia 1.5

Implicit names from identifiers and dot expressions are available as of Julia 1.5.

source

Base.Val — Type.

```
Val(c)
```

Return `Val{c}()`, which contains no run-time data. Types like this can be used to pass the information between functions through the value `c`, which must be an `isbits` value. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

Examples

```
julia> f(::Val{true}) = "Good"
f (generic function with 1 method)

julia> f(::Val{false}) = "Bad"
f (generic function with 2 methods)

julia> f(Val(true))
"Good"
```

source

Core.Vararg – Type.

```
Vararg{T,N}
```

The last parameter of a tuple type `Tuple` can be the special type `Vararg`, which denotes any number of trailing elements. The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `Vararg{T}` corresponds to zero or more elements of type `T`. `Vararg` tuple types are used to represent the arguments accepted by varargs methods (see the section on Varargs Functions in the manual.)

Examples

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

source

Core.Nothing – Type.

```
Nothing
```

A type with no fields that is the type of `nothing`.

source

Base.isnothing – Function.

```
isnothing(x)
```

Return `true` if `x === nothing`, and return `false` if not.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Some – Type.

```
Some{T}
```

A wrapper type used in `Union{Some{T}, Nothing}` to distinguish between the absence of a value (nothing) and the presence of a `nothing` value (i.e. `Some(nothing)`).

Use something to access the value wrapped by a `Some` object.

source

Base.something – Function.

```
something(x, y...)
```

Return the first value in the arguments which is not equal to nothing, if any. Otherwise throw an error. Arguments of type Some are unwrapped.

See also coalesce.

Examples

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

source

Base.Enums.Enum – Type.

```
Enum{T<:Integer}
```

The abstract supertype of all enumerated types defined with `@enum`.

source

`Base.Enums.@enum` — Macro.

```
@enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of `x` and `y`, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

Examples

```
julia> @enum Fruit apple=1 orange=2 kiwi=3

julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
f (generic function with 1 method)

julia> f(apple)
"I'm a Fruit with value: 1"

julia> Fruit(1)
apple::Fruit = 1
```

Values can also be specified inside a `begin` block, e.g.

```
@enum EnumName begin
    value1
    value2
end
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically.

To list all the instances of an enum use `instances`, e.g.

```
julia> instances(Fruit)
(apple, orange, kiwi)
```

source

Core.Expr – Type.

```
Expr(head::Symbol, args...)
```

A type representing compound expressions in parsed julia code (ASTs). Each expression consists of a `head` `Symbol` identifying which kind of expression it is (e.g. a call, for loop, conditional statement, etc.), and subexpressions (e.g. the arguments of a call). The subexpressions are stored in a `Vector{Any}` field called `args`.

See the manual chapter on Metaprogramming and the developer documentation Julia ASTs.

Examples

```
julia> Expr(:call, :+, 1, 2)
:(1 + 2)

julia> dump(:(a ? b : c))
Expr
  head: Symbol if
  args: Array{Any}((3,))
    1: Symbol a
    2: Symbol b
    3: Symbol c
```

source

Core.Symbol – Type.

```
Symbol
```

The type of object used to represent identifiers in parsed julia code (ASTs). Also often used as a name or label to identify an entity (e.g. as a dictionary key). `Symbol`s can be entered using the `:` quote operator:

```
julia> :name
:name

julia> typeof(:name)
Symbol

julia> x = 42
42
```

```
julia> eval(:x)
42
```

`Symbol`s can also be constructed from strings or other values by calling the constructor `Symbol(x...)`.

`Symbol`s are immutable and should be compared using `===`. The implementation re-uses the same object for all `Symbol`s with the same name, so comparison tends to be efficient (it can just compare pointers).

Unlike strings, `Symbol`s are "atomic" or "scalar" entities that do not support iteration over characters.

source

`Core.Symbol` – Method.

```
Symbol(x...) -> Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

Examples

```
julia> Symbol("my", "name")
:myname

julia> Symbol("day", 4)
:day4
```

source

`Core.Module` – Type.

```
Module
```

A `Module` is a separate global variable workspace. See `module` and the manual section about modules for details.

source

45.9 Generic Functions

`Core.Function` – Type.

```
Function
```

Abstract type of all functions.

Examples

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin)

julia> ans <: Function
true
```

source

Base.hasmethod – Function.

```
hasmethod(f, t::Type{<:Tuple}[, kwnames]; world=typemax(UInt)) -> Bool
```

Determine whether the given generic function has a method matching the given `Tuple` of argument types with the upper bound of world age given by `world`.

If a tuple of keyword argument names `kwnames` is provided, this also checks whether the method of `f` matching `t` has the given keyword argument names. If the matching method accepts a variable number of keyword arguments, e.g. with `kwargs...`, any names given in `kwnames` are considered valid. Otherwise the provided names must be a subset of the method's keyword arguments.

See also applicable.

Julia 1.2

Providing keyword argument names requires Julia 1.2 or later.

Examples

```
julia> hasmethod(length, Tuple{Array})
true

julia> hasmethod(sum, Tuple{Function, Array}, (:dims,))
true

julia> hasmethod(sum, Tuple{Function, Array}, (:apples, :bananas))
```

```
false

julia> g(; xs...) = 4;

julia> hasmethod(g, Tuple{}, (:a, :b, :c, :d))  # g accepts arbitrary kwargs
true
```

source

Core.applicable – Function.

```
applicable(f, args...) -> Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

See also hasmethod.

Examples

```
julia> function f(x, y)
           x + y
       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

source

Core.invoke – Function.

```
invoke(f, argtypes::Type, args...; kwargs...)
```

Invoke a method for the given generic function `f` matching the specified types `argtypes` on the specified arguments `args` and passing the keyword arguments `kwargs`. The arguments `args` must conform with the specified types in `argtypes`, i.e. conversion is not automatically performed. This method allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

Be careful when using `invoke` for functions that you don't write. What definition is used for given `argtypes` is an implementation detail unless the function is explicitly states that calling with certain `argtypes` is a part of

public API. For example, the change between `f1` and `f2` in the example below is usually considered compatible because the change is invisible by the caller with a normal (non-`invoke`) call. However, the change is visible if you use `invoke`.

Examples

```
julia> f(x::Real) = x^2;

julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);

julia> f(2)
5

julia> f1(::Integer) = Integer
       f1(::Real) = Real;

julia> f2(x::Real) = _f2(x)
       _f2(::Integer) = Integer
       _f2(_) = Real;

julia> f1(1)
Integer

julia> f2(1)
Integer

julia> invoke(f1, Tuple{Real}, 1)
Real

julia> invoke(f2, Tuple{Real}, 1)
Integer
```

source

`Base.invokelatest` – Function.

```
invokelatest(f, args...; kwargs...)
```

Calls `f(args...; kwargs...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions

of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

source

`new` – Keyword.

```
new
```

Special function available to inner constructors which created a new object of the type. See the manual section on Inner Constructor Methods for more information.

source

`Base.:|>` – Function.

```
|>(x, f)
```

Applies a function to the preceding argument. This allows for easy function chaining.

Examples

```
julia> [1:5;] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

source

`Base.:∘` – Function.

```
f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...)` means `f(g(args...))`. The `∘` symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ<tab>`.

Function composition also works in prefix form: `∘(f, g)` is the same as `f ∘ g`. The prefix form supports composition of multiple functions: `∘(f, g, h) = f ∘ g ∘ h` and splatting `∘(fs...)` for composing an iterable collection of functions.

Julia 1.4

Multiple function composition requires at least Julia 1.4.

Julia 1.5

Composition of one function ⨰(f) requires at least Julia 1.5.

Examples

```
julia> map(uppercase∘first, ["apple", "banana", "carrot"])
3-element Array{Char,1}:
 'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
 'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
 'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> fs = [
           x -> 2x
           x -> x/2
           x -> x-1
           x -> x+1
       ];

julia> ∘(fs...)(3)
3.0
```

source

45.10 Syntax

Core.eval – Function.

```
Core.eval(m::Module, expr)
```

Evaluate an expression in the given module and return the result.

source

Base.MainInclude.eval – Function.

```
eval(expr)
```

Evaluate an expression in the global scope of the containing module. Every `Module` (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

source

Base.@eval – Macro.

```
@eval [mod,] ex
```

Evaluate an expression with values interpolated into it using **eval**. If two arguments are provided, the first is the module to evaluate in.

source

Base.evalfile – Function.

```
evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file using include, evaluate all expressions, and return the value of the last one.

source

Base.esc – Function.

```
esc(e)
```

Only valid in the context of an Expr returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

source

Base.@inbounds – Macro.

```
@inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the in-range check for referencing element **i** of array **A** is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

> Warning
>
> Using **@inbounds** may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually. Only use **@inbounds** when it is certain from the information locally available that all accesses are in bounds.

source

Base.@boundscheck – Macro.

```
@boundscheck(blk)
```

Annotates the expression `blk` as a bounds checking block, allowing it to be elided by `@inbounds`.

> Note
>
> The function in which `@boundscheck` is written must be inlined into its caller in order for `@inbounds` to have effect.

Examples

```
julia> @inline function g(A, i)
           @boundscheck checkbounds(A, i)
           return "accessing ($A)[$i]"
       end;

julia> f1() = return g(1:2, -1);

julia> f2() = @inbounds return g(1:2, -1);

julia> f1()
ERROR: BoundsError: attempt to access 2-element UnitRange{Int64} at index [-1]
Stacktrace:
 [1] throw_boundserror(::UnitRange{Int64}, ::Tuple{Int64}) at ./abstractarray.jl:455
 [2] checkbounds at ./abstractarray.jl:420 [inlined]
 [3] g at ./none:2 [inlined]
 [4] f1() at ./none:1
 [5] top-level scope

julia> f2()
"accessing (1:2)[-1]"
```

> Warning
>
> The `@boundscheck` annotation allows you, as a library writer, to opt-in to allowing other code to remove your bounds checks with `@inbounds`. As noted there, the caller must verify—using information they can access—that their accesses are valid before using `@inbounds`. For indexing into your `AbstractArray` subclasses, for example, this involves checking the indices against its `size`. Therefore,

`@boundscheck` annotations should only be added to a `getindex` or `setindex!` implementation after you are certain its behavior is correct.

source

`Base.@propagate_inbounds` – Macro.

```
@propagate_inbounds
```

Tells the compiler to inline a function while retaining the caller's inbounds context.

source

`Base.@inline` – Macro.

```
@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

```
@inline function bigfunction(x)
    #=
        Function Definition
    =#
end
```

source

`Base.@noinline` – Macro.

```
@noinline
```

Give a hint to the compiler that it should not inline a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented. This is shown in the following example:

```
@noinline function smallfunction(x)
    #=
```

```
        Function Definition
    =#
end

If the function is trivial (for example returning a constant) it might get inlined anyway.
```

source

Base.@nospecialize – Macro.

```
@nospecialize
```

Applied to a function argument name, hints to the compiler that the method should not be specialized for different types of that argument, but instead to use precisely the declared type for each argument. This is only a hint for avoiding excess code generation. Can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the macro must wrap the entire argument expression. When used in a function body, the macro must occur in statement position and before any code.

When used without arguments, it applies to all arguments of the parent scope. In local scope, this means all arguments of the containing function. In global (top-level) scope, this means all methods subsequently defined in the current module.

Specialization can reset back to the default by using @specialize.

```
function example_function(@nospecialize x)
    ...
end

function example_function(@nospecialize(x = 1), y)
    ...
end

function example_function(x, y, z)
    @nospecialize x y
    ...
end

@nospecialize
f(y) = [x for x in y]
@specialize
```

source

Base.@specialize – Macro.

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see @nospecialize.

source

Base.gensym – Function.

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names.

source

Base.@gensym – Macro.

```
@gensym
```

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

source

Base.@goto – Macro.

```
@goto name
```

`@goto name` unconditionally jumps to the statement at the location `@label name`.

`@label` and `@goto` cannot create jumps to different top-level statements. Attempts cause an error. To still use `@goto`, enclose the `@label` and `@goto` in a block.

source

Base.@label – Macro.

```
@label name
```

Labels a statement with the symbolic label `name`. The label marks the end-point of an unconditional jump with `@goto name`.

source

`Base.SimdLoop.@simd` — Macro.

```
@simd
```

Annotate a `for` loop to allow the compiler to take extra liberties to allow loop re-ordering

> Warning
>
> This feature is experimental and could change or disappear in future versions of Julia. Incorrect use of the `@simd` macro may cause unexpected results.

The object iterated over in a `@simd for` loop should be a one-dimensional range. By using `@simd`, you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered, possibly causing different results than without `@simd`.

In many cases, Julia is able to automatically vectorize inner for loops without the use of `@simd`. Using `@simd` gives the compiler a little extra leeway to make it possible in more situations. In either case, your inner loop should have the following properties to allow vectorization:

- The loop must be an innermost loop
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `?:` expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `?:` in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).
- The stride should be unit stride.

> Note
>
> The `@simd` does not assert by default that the loop is completely free of loop-carried memory dependencies, which is an assumption that can easily be violated in generic code. If you are writing non-generic code, you can use `@simd ivdep for ... end` to also assert that:

- There exists no loop-carried memory dependencies

- No iteration ever waits on a previous iteration to make forward progress.

source

Base.@polly – Macro.

```
@polly
```

Tells the compiler to apply the polyhedral optimizer Polly to a function.

source

Base.@generated – Macro.

```
@generated f
@generated(f)
```

`@generated` is used to annotate a function which will be generated. In the body of the generated function, only types of arguments can be read (not the values). The function returns a quoted expression evaluated when the function is called. The `@generated` macro should not be used on functions mutating the global scope or depending on mutable elements.

See Metaprogramming for further details.

Example:

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

source

Base.@pure – Macro.

```
@pure ex
@pure(ex)
```

`@pure` gives the compiler a hint for the definition of a pure function, helping for type inference.

A pure function can only depend on immutable information. This also means a `@pure` function cannot use any global mutable state, including generic functions. Calls to generic functions depend on method tables which are mutable global state. Use with caution, incorrect `@pure` annotation of a function may introduce hard to identify bugs. Double check for calls to generic functions. This macro is intended for internal compiler use and may be subject to changes.

source

Base.@deprecate – Macro.

```
@deprecate old new [ex=true]
```

Deprecate method `old` and specify the replacement call `new`. Prevent `@deprecate` from exporting `old` by setting `ex` to `false`. `@deprecate` defines a new method with the same signature as `old`.

> Julia 1.5
>
> As of Julia 1.5, functions defined by `@deprecate` do not print warning when `julia` is run without the `--depwarn=yes` flag set, as the default value of `--depwarn` option is `no`. The warnings are printed from tests run by `Pkg.test()`.

Examples

```
julia> @deprecate old(x) new(x)
old (generic function with 1 method)

julia> @deprecate old(x) new(x) false
old (generic function with 1 method)
```

source

45.11 Missing Values

Base.Missing – Type.

```
Missing
```

A type with no fields whose singleton instance `missing` is used to represent missing values.

source

`Base.missing` – Constant.

```
missing
```

The singleton instance of type `Missing` representing a missing value.

source

`Base.coalesce` – Function.

```
coalesce(x, y...)
```

Return the first value in the arguments which is not equal to `missing`, if any. Otherwise return `missing`.

See also `something`.

Examples

```
julia> coalesce(missing, 1)
1

julia> coalesce(1, missing)
1

julia> coalesce(nothing, 1)  # returns `nothing`

julia> coalesce(missing, missing)
missing
```

source

`Base.ismissing` – Function.

```
ismissing(x)
```

Indicate whether `x` is `missing`.

source

Base.skipmissing – Function.

```
skipmissing(itr)
```

Return an iterator over the elements in `itr` skipping `missing` values. The returned object can be indexed using indices of `itr` if the latter is indexable. Indices corresponding to missing values are not valid: they are skipped by `keys` and `eachindex`, and a `MissingException` is thrown when trying to use them.

Use `collect` to obtain an `Array` containing the non-`missing` values in `itr`. Note that even if `itr` is a multidimensional array, the result will always be a `Vector` since it is not possible to remove missings while preserving dimensions of the input.

Examples

```
julia> x = skipmissing([1, missing, 2])
skipmissing(Union{Missing, Int64}[1, missing, 2])

julia> sum(x)
3

julia> x[1]
1

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]

julia> argmax(x)
3

julia> collect(keys(x))
2-element Array{Int64,1}:
 1
 3

julia> collect(skipmissing([1, missing, 2]))
2-element Array{Int64,1}:
 1
 2

julia> collect(skipmissing([1 missing; 2 missing]))
```

```
2-element Array{Int64,1}:
 1
 2
```

source

45.12 System

`Base.run` – Function.

```
run(command, args...; wait::Bool = true)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual). Throws an error if anything goes wrong, including the process exiting with a non-zero status (when `wait` is true).

If `wait` is false, the process runs asynchronously. You can later wait for it and check its exit status by calling `success` on the returned process object.

When `wait` is false, the process' I/O streams are directed to `devnull`. When `wait` is true, I/O streams are shared with the parent process. Use `pipeline` to control I/O redirection.

source

`Base.devnull` – Constant.

```
devnull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to `/dev/null` on Unix or `NUL` on Windows. Usage:

```
run(pipeline(`cat test.txt`, devnull))
```

source

`Base.success` – Function.

```
success(command)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual), and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

source

Base.process_running – Function.

```
process_running(p::Process)
```

Determine whether a process is currently running.

source

Base.process_exited – Function.

```
process_exited(p::Process)
```

Determine whether a process has exited.

source

Base.kill – Method.

```
kill(p::Process, signum=Base.SIGTERM)
```

Send a signal to a process. The default is to terminate the process. Returns successfully if the process has already exited, but throws an error if killing the process failed for other reasons (e.g. insufficient permissions).

source

Base.Sys.set_process_title – Function.

```
Sys.set_process_title(title::AbstractString)
```

Set the process title. No-op on some operating systems.

source

Base.Sys.get_process_title – Function.

```
Sys.get_process_title()
```

Get the process title. On some systems, will always return an empty string.

source

Base.ignorestatus – Function.

```
ignorestatus(command)
```

Mark a command object so that running it will not throw an error if the result code is non-zero.

source

Base.detach – Function.

```
detach(command)
```

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

source

Base.Cmd – Type.

```
Cmd(cmd::Cmd; ignorestatus, detach, windows_verbatim, windows_hide, env, dir)
```

Construct a new `Cmd` object, representing an external program and arguments, from `cmd`, while changing the settings of the optional keyword arguments:

- `ignorestatus::Bool`: If `true` (defaults to `false`), then the `Cmd` will not throw an error if the return code is nonzero.
- `detach::Bool`: If `true` (defaults to `false`), then the `Cmd` will be run in a new process group, allowing it to outlive the `julia` process and not have Ctrl-C passed to it.
- `windows_verbatim::Bool`: If `true` (defaults to `false`), then on Windows the `Cmd` will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes " in the command line, and \ or " are preceded by backslashes. `windows_verbatim=true` is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- `windows_hide::Bool`: If `true` (defaults to `false`), then on Windows no new console window is displayed when the `Cmd` is executed. This has no effect if a console is already open or on non-Windows systems.
- `env`: Set environment variables to use when running the `Cmd`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, an array or tuple of `"var"=>val` pairs, or `nothing`. In order to modify (rather than replace) the existing environment, create `env` by `copy(ENV)` and then set `env["var"]=val` as desired.
- `dir::AbstractString`: Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from `cmd` are used. Normally, to create a `Cmd` object in the first place, one uses backticks, e.g.

```
Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

source

Base.setenv – Function.

```
setenv(command::Cmd, env; dir="")
```

Set environment variables to use when running the given `command`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, or zero or more `"var"=>val` pair arguments. In order to modify (rather than replace) the existing environment, create `env` by `copy(ENV)` and then setting `env["var"]=val` as desired, or use `withenv`.

The `dir` keyword argument can be used to specify a working directory for the command.

source

Base.withenv – Function.

```
withenv(f::Function, kv::Pair...)
```

Execute `f` in an environment that is temporarily modified (not replaced as in `setenv`) by zero or more `"var"=>val` arguments `kv`. `withenv` is generally used via the `withenv(kv...) do ... end` syntax. A value of `nothing` can be used to temporarily unset an environment variable (if it is set). When `withenv` returns, the original environment has been restored.

source

Base.pipeline – Method.

```
pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other `pipeline` calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example, `pipeline(a,b,c)` is equivalent to `pipeline(pipeline(a,b),c)`. This provides a more concise way to specify multi-stage pipelines.

Examples:

```
run(pipeline(`ls`, `grep xyz`))
run(pipeline(`ls`, "out.txt"))
run(pipeline("out.txt", `grep xyz`))
```

source

Base.pipeline – Method.

```
pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given `command`. Keyword arguments specify which of the command's streams should be redirected. `append` controls whether file output appends to the file. This is a more general version of the 2-argument `pipeline` function. `pipeline(from, to)` is equivalent to `pipeline(from, stdout=to)` when `from` is a command, and to `pipeline(to, stdin=from)` when `from` is another kind of data source.

Examples:

```
run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))
run(pipeline(`update`, stdout="log.txt", append=true))
```

source

Base.Libc.gethostname – Function.

```
gethostname() -> AbstractString
```

Get the local machine's host name.

source

Base.Libc.getpid – Function.

```
getpid() -> Int32
```

Get Julia's process ID.

source

```
getpid(process) -> Int32
```

Get the child process ID, if it still exists.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Libc.time – Method.

```
time()
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

source

Base.time_ns – Function.

```
time_ns()
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

source

Base.@time – Macro.

```
@time
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

See also @timev, @timed, @elapsed, and @allocated.

> Note
>
> For more serious benchmarking, consider the **@btime** macro from the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

```
julia> @time rand(10^6);
  0.001525 seconds (7 allocations: 7.630 MiB)

julia> @time begin
           sleep(0.3)
           1+1
       end
  0.301395 seconds (8 allocations: 336 bytes)
2
```

source

Base.@timev – Macro.

```
@timev
```

This is a verbose version of the **@time** macro. It first prints the same information as **@time**, then any non-zero memory allocation counters, and then returns the value of the expression.

See also @time, @timed, @elapsed, and @allocated.

```
julia> @timev rand(10^6);
  0.001006 seconds (7 allocations: 7.630 MiB)
elapsed time (ns): 1005567
bytes allocated:   8000256
pool allocs:       6
malloc() calls:    1
```

source

Base.@timed – Macro.

```
@timed
```

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

See also @time, @timev, @elapsed, and @allocated.

```
julia> stats = @timed rand(10^6);

julia> stats.time
0.006634834

julia> stats.bytes
8000256

julia> stats.gctime
0.0055765

julia> propertynames(stats.gcstats)
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time, :pause, :full_sweep)

julia> stats.gcstats.total_time
5576500
```

Julia 1.5

The return type of this macro was changed from `Tuple` to `NamedTuple` in Julia 1.5.

source

Base.@elapsed – Macro.

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

See also @time, @timev, @timed, and @allocated.

```
julia> @elapsed sleep(0.3)
0.301391426
```

source

Base.@allocated – Macro.

```
@allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

See also @time, @timev, @timed, and @elapsed.

```
julia> @allocated rand(10^6)
8000080
```

source

Base.EnvDict – Type.

```
EnvDict() -> EnvDict
```

A singleton of this type provides a hash table interface to environment variables.

source

Base.ENV – Constant.

```
ENV
```

Reference to the singleton `EnvDict`, providing a dictionary interface to system environment variables.

(On Windows, system environment variables are case-insensitive, and `ENV` correspondingly converts all keys to uppercase for display, iteration, and copying. Portable code should not rely on the ability to distinguish variables by case, and should beware that setting an ostensibly lowercase variable may result in an uppercase `ENV` key.)

source

`Base.Sys.isunix` – Function.

```
Sys.isunix([os])
```

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

source

`Base.Sys.isapple` – Function.

```
Sys.isapple([os])
```

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

source

`Base.Sys.islinux` – Function.

```
Sys.islinux([os])
```

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

source

`Base.Sys.isbsd` – Function.

```
Sys.isbsd([os])
```

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

Note

The Darwin kernel descends from BSD, which means that `Sys.isbsd()` is `true` on macOS systems. To exclude macOS from a predicate, use `Sys.isbsd() && !Sys.isapple()`.

source

`Base.Sys.isfreebsd` – Function.

```
Sys.isfreebsd([os])
```

Predicate for testing if the OS is a derivative of FreeBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on FreeBSD but also on other BSD-based systems. `Sys.isfreebsd()` refers only to FreeBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

`Base.Sys.isopenbsd` – Function.

```
Sys.isopenbsd([os])
```

Predicate for testing if the OS is a derivative of OpenBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on OpenBSD but also on other BSD-based systems. `Sys.isopenbsd()` refers only to OpenBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

`Base.Sys.isnetbsd` – Function.

```
Sys.isnetbsd([os])
```

Predicate for testing if the OS is a derivative of NetBSD. See documentation in Handling Operating System Variation.

> Note
>
> Not to be confused with `Sys.isbsd()`, which is `true` on NetBSD but also on other BSD-based systems. `Sys.isnetbsd()` refers only to NetBSD.
>
> Julia 1.1
>
> This function requires at least Julia 1.1.

source

`Base.Sys.isdragonfly` – Function.

```
Sys.isdragonfly([os])
```

Predicate for testing if the OS is a derivative of DragonFly BSD. See documentation in Handling Operating System Variation.

> Note
>
> Not to be confused with `Sys.isbsd()`, which is `true` on DragonFly but also on other BSD-based systems. `Sys.isdragonfly()` refers only to DragonFly.
>
> Julia 1.1
>
> This function requires at least Julia 1.1.

source

`Base.Sys.iswindows` – Function.

```
Sys.iswindows([os])
```

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

source

`Base.Sys.windows_version` – Function.

```
Sys.windows_version()
```

Return the version number for the Windows NT Kernel as a `VersionNumber`, i.e. `v"major.minor.build"`, or `v"0.0.0"` if this is not running on Windows.

source

`Base.Sys.free_memory` – Function.

```
Sys.free_memory()
```

Get the total free memory in RAM in bytes.

source

`Base.Sys.total_memory` – Function.

```
Sys.total_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes.

source

`Base.@static` – Macro.

```
@static
```

Partially evaluate an expression at parse time.

For example, `@static Sys.iswindows() ? foo : bar` will evaluate `Sys.iswindows()` and insert either `foo` or `bar` into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a `ccall` to a non-existent function. `@static if Sys.isapple() foo end` and `@static foo <&&,¦¦> bar` are also valid syntax.

source

45.13 Versioning

`Base.VersionNumber` – Type.

```
VersionNumber
```

Version number type which follow the specifications of semantic versioning, composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. See also `@v_str`.

Examples

```
julia> VersionNumber("1.2.3")
v"1.2.3"

julia> VersionNumber("2.0.1-rc1")
v"2.0.1-rc1"
```

source

`Base.@v_str` – Macro.

```
@v_str
```

String macro used to parse a string to a `VersionNumber`.

Examples

```
julia> v"1.2.3"
v"1.2.3"

julia> v"2.0.1-rc1"
v"2.0.1-rc1"
```

source

45.14 Errors

`Base.error` – Function.

```
error(message::AbstractString)
```

Raise an `ErrorException` with the given message.

source

```
error(msg...)
```

Raise an `ErrorException` with the given message.

source

Core.throw – Function.

```
throw(e)
```

Throw an object as an exception.

source

Base.rethrow – Function.

```
rethrow()
```

Rethrow the current exception from within a `catch` block. The rethrown exception will continue propagation as if it had not been caught.

> Note
>
> The alternative form `rethrow(e)` allows you to associate an alternative exception object `e` with the current backtrace. However this misrepresents the program state at the time of the error so you're encouraged to instead throw a new exception using `throw(e)`. In Julia 1.1 and above, using `throw(e)` will preserve the root cause exception on the stack, as described in catch_stack.

source

Base.backtrace – Function.

```
backtrace()
```

Get a backtrace object for the current program point.

source

Base.catch_backtrace – Function.

```
catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

source

Base.catch_stack – Function.

```
catch_stack(task=current_task(); [inclue_bt=true])
```

Get the stack of exceptions currently being handled. For nested catch blocks there may be more than one current exception in which case the most recently thrown exception is last in the stack. The stack is returned as a Vector of `(exception,backtrace)` pairs, or a Vector of exceptions if `include_bt` is false.

Explicitly passing `task` will return the current exception stack on an arbitrary task. This is useful for inspecting tasks which have failed due to uncaught exceptions.

> Julia 1.1
>
> This function is experimental in Julia 1.1 and will likely be renamed in a future release (see https://github.com/JuliaLang/julia/pull/29901).

source

`Base.@assert` – Macro.

```
@assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. Preferred syntax for writing assertions. Message `text` is optionally displayed upon assertion failure.

> Warning
>
> An assert might be disabled at various optimization levels. Assert should therefore only be used as a debugging tool and not used for authentication verification (e.g., verifying passwords), nor should side effects needed for the function to work correctly be used inside of asserts.

Examples

```
julia> @assert iseven(3) "3 is an odd number!"
ERROR: AssertionError: 3 is an odd number!

julia> @assert isodd(3) "What even are numbers?"
```

source

`Core.ArgumentError` – Type.

```
ArgumentError(msg)
```

The parameters to a function call do not match a valid signature. Argument `msg` is a descriptive error string.

source

Core.AssertionError — Type.

```
AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

Examples

```
julia> @assert false "this is not true"
ERROR: AssertionError: this is not true
```

`AssertionError` is usually thrown from `@assert`.

source

Core.BoundsError — Type.

```
BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element at index `i`.

Examples

```
julia> A = fill(1.0, 7);

julia> A[8]
ERROR: BoundsError: attempt to access 7-element Array{Float64,1} at index [8]
Stacktrace:
 [1] getindex(::Array{Float64,1}, ::Int64) at ./array.jl:660
 [2] top-level scope

julia> B = fill(1.0, (2,3));

julia> B[2, 4]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [2, 4]
Stacktrace:
 [1] getindex(::Array{Float64,2}, ::Int64, ::Int64) at ./array.jl:661
 [2] top-level scope

julia> B[9]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [9]
Stacktrace:
```

```
[1] getindex(::Array{Float64,2}, ::Int64) at ./array.jl:660
[2] top-level scope
```

source

Base.CompositeException – Type.

```
CompositeException
```

Wrap a `Vector` of exceptions thrown by a `Task` (e.g. generated from a remote worker over a channel or an asynchronously executing local I/O write or a remote worker under `pmap`) with information about the series of exceptions. For example, if a group of workers are executing several tasks, and multiple workers fail, the resulting `CompositeException` will contain a "bundle" of information from each worker indicating where and why the exception(s) occurred.

source

Base.DimensionMismatch – Type.

```
DimensionMismatch([msg])
```

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

source

Core.DivideError – Type.

```
DivideError()
```

Integer division was attempted with a denominator value of 0.

Examples

```
julia> 2/0
Inf

julia> div(2, 0)
ERROR: DivideError: integer division error
Stacktrace:
[...]
```

source

Core.DomainError – Type.

```
DomainError(val)
DomainError(val, msg)
```

The argument `val` to a function or constructor is outside the valid domain.

Examples

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

source

Base.EOFError – Type.

```
EOFError()
```

No more data was available to read from a file or stream.

source

Core.ErrorException – Type.

```
ErrorException(msg)
```

Generic error type. The error message, in the `.msg` field, may provide more specific details.

Examples

```
julia> ex = ErrorException("I've done a bad thing");

julia> ex.msg
"I've done a bad thing"
```

source

Core.InexactError – Type.

```
InexactError(name::Symbol, T, val)
```

Cannot exactly convert `val` to type `T` in a method of function `name`.

Examples

```
julia> convert(Float64, 1+2im)
ERROR: InexactError: Float64(1 + 2im)
Stacktrace:
[...]
```

source

`Core.InterruptException` — Type.

```
InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

Note that, in Julia script started without `-i` (interactive) option, `InterruptException` is not thrown by default. Calling `Base.exit_on_sigint(false)` in the script can recover the behavior of the REPL. Alternatively, a Julia script can be started with

```
julia -e "include(popfirst!(ARGS))" script.jl
```

to let `InterruptException` be thrown by CTRL+C during the execution.

source

`Base.KeyError` — Type.

```
KeyError(key)
```

An indexing operation into an `AbstractDict` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

source

`Core.LoadError` — Type.

```
LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `include`ing, `require`ing, or `using` a file. The error specifics should be available in the `.error` field.

source

Core.MethodError — Type.

```
MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

source

Base.MissingException — Type.

```
MissingException(msg)
```

Exception thrown when a missing value is encountered in a situation where it is not supported. The error message, in the **msg** field may provide more specific details.

source

Core.OutOfMemoryError — Type.

```
OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

source

Core.ReadOnlyMemoryError — Type.

```
ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

source

Core.OverflowError — Type.

```
OverflowError(msg)
```

The result of an expression is too large for the specified type and will cause a wraparound.

source

Base.ProcessFailedException — Type.

```
ProcessFailedException
```

Indicates problematic exit status of a process. When running commands or pipelines, this is thrown to indicate a nonzero exit code was returned (i.e. that the invoked process failed).

source

Core.StackOverflowError – Type.

```
StackOverflowError()
```

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

source

Base.SystemError – Type.

```
SystemError(prefix::AbstractString, [errno::Int32])
```

A system call failed with an error code (in the `errno` global variable).

source

Core.TypeError – Type.

```
TypeError(func::Symbol, context::AbstractString, expected::Type, got)
```

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

source

Core.UndefKeywordError – Type.

```
UndefKeywordError(var::Symbol)
```

The required keyword argument `var` was not assigned in a function call.

Examples

```
julia> function my_func(;my_arg)
           return my_arg + 1
       end
my_func (generic function with 1 method)

julia> my_func()
ERROR: UndefKeywordError: keyword argument my_arg not assigned
```

```
Stacktrace:
 [1] my_func() at ./REPL[1]:2
 [2] top-level scope at REPL[2]:1
```

source

Core.UndefRefError – Type.

```
UndefRefError()
```

The item or field is not defined for the given object.

Examples

```
julia> struct MyType
           a::Vector{Int}
           MyType() = new()
       end

julia> A = MyType()
MyType(#undef)

julia> A.a
ERROR: UndefRefError: access to undefined reference
Stacktrace:
[...]
```

source

Core.UndefVarError – Type.

```
UndefVarError(var::Symbol)
```

A symbol in the current scope is not defined.

Examples

```
julia> a
ERROR: UndefVarError: a not defined

julia> a = 1;
```

```
julia> a
1
```

source

Base.StringIndexError – Type.

```
StringIndexError(str, i)
```

An error occurred when trying to access `str` at index `i` that is not valid.

source

Core.InitError – Type.

```
InitError(mod::Symbol, error)
```

An error occurred when running a module's `__init__` function. The actual error thrown is available in the `.error` field.

source

Base.retry – Function.

```
retry(f;  delays=ExponentialBackOff(), check=nothing) -> Function
```

Return an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time `check` returns `true`, after waiting the number of seconds specified in `delays`. `check` should input `delays`'s current state and the `Exception`.

> Julia 1.2
>
> Before Julia 1.2 this signature was restricted to `f::Function`.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5, max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, IOError))(io, 128; all=false)
```

source

Base.ExponentialBackOff — Type.

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0, factor=5.0, jitter=0.1)
```

A Float64 iterator of length n whose elements exponentially increase at a rate in the interval factor * (1 ± jitter). The first element is first_delay and all elements are clamped to max_delay.

source

45.15 Events

Base.Timer — Method.

```
Timer(callback::Function, delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling wait on the timer object) and calls the function callback.

Waiting tasks are woken and the function callback is called after an initial delay of delay seconds, and then repeating with the given interval in seconds. If interval is equal to 0, the timer is only triggered once. The function callback is called with a single argument, the timer itself. When the timer is closed (by close waiting tasks are woken with an error. Use isopen to check whether a timer is still active.

Examples

Here the first number is printed after a delay of two seconds, then the following numbers are printed quickly.

```
julia> begin
           i = 0
           cb(timer) = (global i += 1; println(i))
           t = Timer(cb, 2, interval=0.2)
           wait(t)
           sleep(0.5)
           close(t)
       end
1
2
3
```

source

Base.Timer — Type.

```
Timer(delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object).

Waiting tasks are woken after an initial delay of `delay` seconds, and then repeating with the given `interval` in seconds. If `interval` is equal to `0`, the timer is only triggered once. When the timer is closed (by `close` waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

source

`Base.AsyncCondition` – Type.

```
AsyncCondition()
```

Create a async condition that wakes up tasks waiting for it (by calling `wait` on the object) when notified from C by a call to `uv_async_send`. Waiting tasks are woken with an error when the object is closed (by `close`. Use `isopen` to check whether it is still active.

source

`Base.AsyncCondition` – Method.

```
AsyncCondition(callback::Function)
```

Create a async condition that calls the given `callback` function. The `callback` is passed one argument, the async condition object itself.

source

45.16 Reflection

`Base.nameof` – Method.

```
nameof(m::Module) -> Symbol
```

Get the name of a `Module` as a `Symbol`.

Examples

```
julia> nameof(Base.Broadcast)
:Broadcast
```

source

Base.parentmodule – Function.

```
parentmodule(m::Module) -> Module
```

Get a module's enclosing `Module`. `Main` is its own parent.

Examples

```
julia> parentmodule(Main)
Main

julia> parentmodule(Base.Broadcast)
Base
```

source

```
parentmodule(t::DataType) -> Module
```

Determine the module containing the definition of a (potentially `UnionAll`-wrapped) `DataType`.

Examples

```
julia> module Foo
           struct Int end
       end
Foo

julia> parentmodule(Int)
Core

julia> parentmodule(Foo.Int)
Foo
```

source

```
parentmodule(f::Function) -> Module
```

Determine the module containing the (first) definition of a generic function.

source

```
parentmodule(f::Function, types) -> Module
```

Determine the module containing a given definition of a generic function.

source

Base.pathof – Method.

```
pathof(m::Module)
```

Return the path of the `m.jl` file that was used to `import` module `m`, or `nothing` if `m` was not imported from a package.

Use dirname to get the directory part and basename to get the file name part of the path.

source

Base.moduleroot – Function.

```
moduleroot(m::Module) -> Module
```

Find the root module of a given module. This is the first module in the chain of parent modules of `m` which is either a registered root module or which is its own parent module.

source

Base.@__MODULE__ – Macro.

```
@__MODULE__ -> Module
```

Get the `Module` of the toplevel eval, which is the `Module` code is currently being read from.

source

Base.fullname – Function.

```
fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

```
julia> fullname(Base.Iterators)
(:Base, :Iterators)

julia> fullname(Main)
(:Main,)
```

source

Base.names – Function.

```
names(x::Module; all::Bool = false, imported::Bool = false)
```

Get an array of the names exported by a `Module`, excluding deprecated names. If `all` is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If `imported` is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in `Main` are considered "exported", since it is not idiomatic to explicitly export names from `Main`.

source

Core.nfields – Function.

```
nfields(x) -> Int
```

Get the number of fields in the given object.

Examples

```
julia> a = 1//2;

julia> nfields(a)
2

julia> b = 1
1

julia> nfields(b)
0

julia> ex = ErrorException("I've done a bad thing");

julia> nfields(ex)
1
```

In these examples, `a` is a Rational, which has two fields. `b` is an `Int`, which is a primitive bitstype with no fields at all. `ex` is an ErrorException, which has one field.

source

Base.isconst – Function.

```
isconst(m::Module, s::Symbol) -> Bool
```

Determine whether a global is declared `const` in a given `Module`.

source

Base.nameof – Method.

```
nameof(f::Function) -> Symbol
```

Get the name of a generic `Function` as a symbol. For anonymous functions, this is a compiler-generated name. For explicitly-declared subtypes of `Function`, it is the name of the function's type.

source

Base.functionloc – Method.

```
functionloc(f::Function, types)
```

Returns a tuple `(filename,line)` giving the location of a generic `Function` definition.

source

Base.functionloc – Method.

```
functionloc(m::Method)
```

Returns a tuple `(filename,line)` giving the location of a `Method` definition.

source

45.17 Internals

Base.GC.gc – Function.

```
GC.gc([full=true])
```

Perform garbage collection. The argument `full` determines the kind of collection: A full collection (default) sweeps all objects, which makes the next GC scan much slower, while an incremental collection may only sweep so-called young objects.

Warning

Excessive use will likely lead to poor performance.

source

Base.GC.enable – Function.

```
GC.enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Return previous GC state.

Warning

Disabling garbage collection should be used only with caution, as it can cause memory use to grow without bound.

source

Base.GC.@preserve – Macro.

```
GC.@preserve x1 x2 ... xn expr
```

Mark the objects `x1, x2, ...` as being in use during the evaluation of the expression `expr`. This is only required in unsafe code where `expr` implicitly uses memory or other resources owned by one of the `x`s.

Implicit use of `x` covers any indirect use of resources logically owned by `x` which the compiler cannot see. Some examples:

- Accessing memory of an object directly via a `Ptr`
- Passing a pointer to `x` to `ccall`
- Using resources of `x` which would be cleaned up in the finalizer.

`@preserve` should generally not have any performance impact in typical use cases where it briefly extends object lifetime. In implementation, `@preserve` has effects such as protecting dynamically allocated objects from garbage collection.

Examples

When loading from a pointer with `unsafe_load`, the underlying object is implicitly used, for example `x` is implicitly used by `unsafe_load(p)` in the following:

```
julia> let
           x = Ref{Int}(101)
           p = Base.unsafe_convert(Ptr{Int}, x)
           GC.@preserve x unsafe_load(p)
       end
101
```

When passing pointers to `ccall`, the pointed-to object is implicitly used and should be preserved. (Note however that you should normally just pass `x` directly to `ccall` which counts as an explicit use.)

```
julia> let
           x = "Hello"
           p = pointer(x)
           Int(GC.@preserve x @ccall strlen(p::Cstring)::Csize_t)
           # Preferred alternative
           Int(@ccall strlen(x::Cstring)::Csize_t)
       end
5
```

source

Base.Meta.lower – Function.

```
lower(m, x)
```

Takes the expression `x` and returns an equivalent expression in lowered form for executing in module `m`. See also code_lowered.

source

Base.Meta.@lower – Macro.

```
@lower [m] x
```

Return lowered form of the expression `x` in module `m`. By default `m` is the module in which the macro is called. See also lower.

source

Base.Meta.parse – Method.

```
parse(str, start; greedy=true, raise=true, depwarn=true)
```

Parse the expression string and return an expression (which could later be passed to eval for execution). `start` is the index of the first character to start parsing. If `greedy` is `true` (default), `parse` will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return `Expr(:incomplete, "(error message)")`. If `raise` is `true` (default), syntax errors other than incomplete expressions will raise an error. If `raise` is `false`, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed.

```
julia> Meta.parse("x = 3, y = 5", 7)
(:(y = 5), 13)

julia> Meta.parse("x = 3, y = 5", 5)
(:((3, y) = 5), 13)
```

source

`Base.Meta.parse` – Method.

```
parse(str; raise=true, depwarn=true)
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `raise` is `true` (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed.

```
julia> Meta.parse("x = 3")
:(x = 3)

julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))

julia> Meta.parse("1.0.2")
ERROR: Base.Meta.ParseError("invalid numeric constant \"1.0.\"")
Stacktrace:
[...]

julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant \"1.0.\"")))
```

source

`Base.Meta.ParseError` – Type.

```
ParseError(msg)
```

The expression passed to the parse function could not be interpreted as a valid Julia expression.

source

Core.QuoteNode – Type.

```
QuoteNode
```

A quoted piece of code, that does not support interpolation. See the manual section about QuoteNodes for details.

source

Base.macroexpand – Function.

```
macroexpand(m::Module, x; recursive=true)
```

Take the expression x and return an equivalent expression with all macros removed (expanded) for executing in module m. The recursive keyword controls whether deeper levels of nested macros are also expanded. This is demonstrated in the example below:

```
julia> module M
           macro m1()
               42
           end
           macro m2()
               :(@m1())
           end
       end
M

julia> macroexpand(M, :(@m2()), recursive=true)
42

julia> macroexpand(M, :(@m2()), recursive=false)
:(#= REPL[16]:6 =# M.@m1)
```

source

Base.@macroexpand – Macro.

```
@macroexpand
```

Return equivalent expression with all macros removed (expanded).

There are differences between **@macroexpand** and macroexpand.

- While macroexpand takes a keyword argument **recursive**, **@macroexpand** is always recursive. For a non recursive macro version, see @macroexpand1.
- While macroexpand has an explicit **module** argument, **@macroexpand** always expands with respect to the module in which it is called.

This is best seen in the following example:

```
julia> module M
           macro m()
               1
           end
           function f()
               (@macroexpand(@m),
                macroexpand(M, :(@m)),
                macroexpand(Main, :(@m))
               )
           end
       end
M

julia> macro m()
           2
       end
@m (macro with 1 method)

julia> M.f()
(1, 1, 2)
```

With **@macroexpand** the expression expands where **@macroexpand** appears in the code (module **M** in the example). With **macroexpand** the expression expands in the module given as the first argument.

source

Base.@macroexpand1 – Macro.

```
@macroexpand1
```

Non recursive version of `@macroexpand`.

source

`Base.code_lowered` – Function.

```
code_lowered(f, types; generated=true, debuginfo=:default)
```

Return an array of the lowered forms (IR) for the methods matching the given generic function and type signature.

If `generated` is `false`, the returned `CodeInfo` instances will correspond to fallback implementations. An error is thrown if no fallback implementation exists. If `generated` is `true`, these `CodeInfo` instances will correspond to the method bodies yielded by expanding the generators.

The keyword `debuginfo` controls the amount of code metadata present in the output.

Note that an error will be thrown if `types` are not leaf types when `generated` is `true` and any of the corresponding methods are an `@generated` method.

source

`Base.code_typed` – Function.

```
code_typed(f, types; optimize=true, debuginfo=:default)
```

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature. The keyword argument `optimize` controls whether additional optimizations, such as inlining, are also applied. The keyword `debuginfo` controls the amount of code metadata present in the output, possible options are `:source` or `:none`.

source

`Base.precompile` – Function.

```
precompile(f, args::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

source

45.18 Meta

`Base.Meta.quot` – Function.

```
Meta.quot(ex)::Expr
```

Quote expression `ex` to produce an expression with head `quote`. This can for instance be used to represent objects of type `Expr` in the AST. See also the manual section about QuoteNode.

Examples

```
julia> eval(Meta.quot(:x))
:x

julia> dump(Meta.quot(:x))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Symbol x

julia> eval(Meta.quot(:(1+2)))
:(1 + 2)
```

source

`Base.Meta.isexpr` – Function.

```
Meta.isexpr(ex, head[, n])::Bool
```

Check if `ex` is an expression with head `head` and `n` arguments.

Examples

```
julia> ex = :(f(x))
:(f(x))

julia> Meta.isexpr(ex, :block)
false

julia> Meta.isexpr(ex, :call)
true
```

```
julia> Meta.isexpr(ex, [:block, :call]) # multiple possible heads
true

julia> Meta.isexpr(ex, :call, 1)
false

julia> Meta.isexpr(ex, :call, 2)
true
```

source

Base.Meta.show_sexpr – Function.

```
Meta.show_sexpr([io::IO,], ex)
```

Show expression `ex` as a lisp style S-expression.

Examples

```
julia> Meta.show_sexpr(:(f(x, g(y,z))))
(:call, :f, :x, (:call, :g, :y, :z))
```

source

Chapter 46

Collections and Data Structures

46.1 Iteration

Sequential iteration is implemented by the `iterate` function. The general `for` loop:

```
for i in iter   # or  "for i = iter"
    # body
end
```

is translated into:

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type. See the manual section on the iteration interface for more details about defining a custom iterable type.

`Base.iterate` – Function.

```
iterate(iter [, state]) -> Union{Nothing, Tuple{Any, Any}}
```

Advance the iterator to obtain the next element. If no elements remain, `nothing` should be returned. Otherwise, a 2-tuple of the next element and the new iteration state should be returned.

source

Base.IteratorSize — Type.

```
IteratorSize(itertype::Type) -> IteratorSize
```

Given the type of an iterator, return one of the following values:

- `SizeUnknown()` if the length (number of elements) cannot be determined in advance.
- `HasLength()` if there is a fixed, finite length.
- `HasShape{N}()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case `N` should give the number of dimensions, and the `axes` function is valid for the iterator.
- `IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`. This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.IteratorSize(1:5)
Base.HasShape{1}()

julia> Base.IteratorSize((2,3))
Base.HasLength()
```

source

Base.IteratorEltype — Type.

```
IteratorEltype(itertype::Type) -> IteratorEltype
```

Given the type of an iterator, return one of the following values:

- `EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.
- `HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.IteratorEltype(1:5)
Base.HasEltype()
```

source

Fully implemented by:

- AbstractRange
- UnitRange
- Tuple
- Number
- AbstractArray
- BitSet
- IdDict
- Dict
- WeakKeyDict
- EachLine
- AbstractString
- Set
- Pair
- NamedTuple

46.2 Constructors and Types

Base.AbstractRange – Type.

```
AbstractRange{T}
```

Supertype for ranges with elements of type T. UnitRange and other types are subtypes of this.

source

Base.OrdinalRange – Type.

```
OrdinalRange{T, S} <: AbstractRange{T}
```

Supertype for ordinal ranges with elements of type `T` with spacing(s) of type `S`. The steps should be always-exact multiples of `oneunit`, and `T` should be a "discrete" type, which cannot have values smaller than `oneunit`. For example, `Integer` or `Date` types would qualify, whereas `Float64` would not (since this type can represent values smaller than `oneunit(Float64)`. `UnitRange`, `StepRange`, and other types are subtypes of this.

source

`Base.AbstractUnitRange` — Type.

```
AbstractUnitRange{T} <: OrdinalRange{T, T}
```

Supertype for ranges with a step size of `oneunit(T)` with elements of type `T`. `UnitRange` and other types are subtypes of this.

source

`Base.StepRange` — Type.

```
StepRange{T, S} <: OrdinalRange{T, S}
```

Ranges with elements of type `T` with spacing of type `S`. The step between each element is constant, and the range is defined in terms of a `start` and `stop` of type `T` and a `step` of type `S`. Neither `T` nor `S` should be floating point types. The syntax `a:b:c` with `b > 1` and `a`, `b`, and `c` all integers creates a `StepRange`.

Examples

```
julia> collect(StepRange(1, Int8(2), 10))
5-element Array{Int64,1}:
 1
 3
 5
 7
 9

julia> typeof(StepRange(1, Int8(2), 10))
StepRange{Int64,Int8}

julia> typeof(1:3:6)
StepRange{Int64,Int64}
```

source

Base.UnitRange – Type.

```
UnitRange{T<:Real}
```

A range parameterized by a `start` and `stop` of type `T`, filled with elements spaced by `1` from `start` until `stop` is exceeded. The syntax `a:b` with `a` and `b` both `Integer`s creates a `UnitRange`.

Examples

```
julia> collect(UnitRange(2.3, 5.2))
3-element Array{Float64,1}:
 2.3
 3.3
 4.3

julia> typeof(1:10)
UnitRange{Int64}
```

source

Base.LinRange – Type.

```
LinRange{T}
```

A range with `len` linearly spaced elements between its `start` and `stop`. The size of the spacing is controlled by `len`, which must be an `Int`.

Examples

```
julia> LinRange(1.5, 5.5, 9)
9-element LinRange{Float64}:
 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5
```

Compared to using range, directly constructing a `LinRange` should have less overhead but won't try to correct for floating point errors:

```
julia> collect(range(-0.1, 0.3, length=5))
5-element Array{Float64,1}:
 -0.1
  0.0
```

```
 0.1
 0.2
 0.3

julia> collect(LinRange(-0.1, 0.3, 5))
5-element Array{Float64,1}:
 -0.1
 -1.3877787807814457e-17
  0.09999999999999999
  0.19999999999999998
  0.3
```

source

46.3 General Collections

Base.isempty – Function.

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

Examples

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

source

```
isempty(condition)
```

Return `true` if no tasks are waiting on the condition, `false` otherwise.

source

Base.empty! – Function.

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> empty!(A);

julia> A
Dict{String,Int64}()
```

source

`Base.length` – Function.

```
length(collection) -> Integer
```

Return the number of elements in the collection.

Use `lastindex` to get the last valid index of an indexable collection.

Examples

```
julia> length(1:5)
5

julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

Fully implemented by:

- `AbstractRange`
- `UnitRange`

- Tuple
- Number
- AbstractArray
- BitSet
- IdDict
- Dict
- WeakKeyDict
- AbstractString
- Set
- NamedTuple

46.4 Iterable Collections

Base.in — Function.

```
in(item, collection) -> Bool
∈(item, collection) -> Bool
∋(collection, item) -> Bool
```

Determine whether an item is in the given collection, in the sense that it is == to one of the values generated by iterating over the collection. Returns a `Bool` value, except if `item` is missing or `collection` contains `missing` but not `item`, in which case `missing` is returned (three-valued logic, matching the behavior of any and ==).

Some collections follow a slightly different definition. For example, Sets check whether the item isequal to one of the elements. Dicts look for `key=>value` pairs, and the key is compared using isequal. To test for the presence of a key in a dictionary, use haskey or `k in keys(dict)`. For these collections, the result is always a `Bool` and never `missing`.

To determine whether an item is not in a given collection, see :∉. You may also negate the `in` by doing `!(a in b)` which is logically similar to "not in".

When broadcasting with `in.(items, collection)` or `items .∈ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection `items` is `in` the value at the corresponding

position in `collection`. To get a vector indicating whether each value in `items` is in `collection`, wrap `collection` in a tuple or a `Ref` like this: `in.(items, Ref(collection))` or `items .∈ Ref(collection)`.

Examples

```
julia> a = 1:3:20
1:3:19

julia> 4 in a
true

julia> 5 in a
false

julia> missing in [1, 2]
missing

julia> 1 in [2, missing]
missing

julia> 1 in [1, missing]
true

julia> missing in Set([1, 2])
false

julia> !(21 in a)
true

julia> !(19 in a)
false

julia> [1, 2] .∈ [2, 3]
2-element BitArray{1}:
 0
 0

julia> [1, 2] .∈ ([2, 3],)
2-element BitArray{1}:
 0
```

```
1
```

source

Base.:∉ – Function.

```
∉(item, collection) -> Bool
∌(collection, item) -> Bool
```

Negation of ∈ and ∋, i.e. checks that `item` is not in `collection`.

When broadcasting with `items .∉ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection `items` is not in the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is not in `collection`, wrap `collection` in a tuple or a `Ref` like this: `items .∉ Ref(collection)`.

Examples

```
julia> 1 ∉ 2:4
true

julia> 1 ∉ 1:3
false

julia> [1, 2] .∉ [2, 3]
2-element BitArray{1}:
 1
 1

julia> [1, 2] .∉ ([2, 3],)
2-element BitArray{1}:
 1
 0
```

source

Base.eltype – Function.

```
eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given `type`. For dictionary types, this will be a `Pair{KeyType,ValType}`. The definition `eltype(x) = eltype(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

Examples

```
julia> eltype(fill(1f0, (2,2)))
Float32

julia> eltype(fill(0x1, (2,2)))
UInt8
```

source

Base.indexin – Function.

```
indexin(a, b)
```

Return an array containing the first index in `b` for each value in `a` that is a member of `b`. The output array contains `nothing` wherever `a` is not a member of `b`.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];

julia> b = ['a', 'b', 'c'];

julia> indexin(a, b)
6-element Array{Union{Nothing, Int64},1}:
 1
 2
 3
 2
  nothing
 1

julia> indexin(b, a)
3-element Array{Union{Nothing, Int64},1}:
 1
 2
 3
```

source

Base.unique – Function.

```
unique(itr)
```

Return an array containing only the unique elements of collection `itr`, as determined by `isequal`, in the order that the first of each set of equivalent elements originally appears. The element type of the input is preserved.

Examples

```
julia> unique([1, 2, 6, 2])
3-element Array{Int64,1}:
 1
 2
 6

julia> unique(Real[1, 1.0, 2])
2-element Array{Real,1}:
 1
 2
```

source

```
unique(f, itr)
```

Returns an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

Examples

```
julia> unique(x -> x^2, [1, -1, 3, -3, 4])
3-element Array{Int64,1}:
 1
 3
 4
```

source

```
unique(A::AbstractArray; dims::Int)
```

Return unique regions of `A` along dimension `dims`.

Examples

```
julia> A = map(isodd, reshape(Vector(1:8), (2,2,2)))
2×2×2 Array{Bool,3}:
[:, :, 1] =
 1  1
 0  0

[:, :, 2] =
 1  1
 0  0

julia> unique(A)
2-element Array{Bool,1}:
 1
 0

julia> unique(A, dims=2)
2×1×2 Array{Bool,3}:
[:, :, 1] =
 1
 0

[:, :, 2] =
 1
 0

julia> unique(A, dims=3)
2×2×1 Array{Bool,3}:
[:, :, 1] =
 1  1
 0  0
```

source

`Base.unique!` – Function.

```
unique!(f, A::AbstractVector)
```

Selects one value from `A` for each unique value produced by `f` applied to elements of `A` , then return the modified A.

Julia 1.1

This method is available as of Julia 1.1.

Examples

```
julia> unique!(x -> x^2, [1, -1, 3, -3, 4])
3-element Array{Int64,1}:
 1
 3
 4

julia> unique!(n -> n%3, [5, 1, 8, 9, 3, 4, 10, 7, 2, 6])
3-element Array{Int64,1}:
 5
 1
 9

julia> unique!(iseven, [2, 3, 5, 7, 9])
2-element Array{Int64,1}:
 2
 3
```

source

```
unique!(A::AbstractVector)
```

Remove duplicate items as determined by isequal, then return the modified A. unique! will return the elements of A in the order that they occur. If you do not care about the order of the returned data, then calling (sort!(A); unique!(A)) will be much more efficient as long as the elements of A can be sorted.

Examples

```
julia> unique!([1, 1, 1])
1-element Array{Int64,1}:
 1

julia> A = [7, 3, 2, 3, 7, 5];

julia> unique!(A)
4-element Array{Int64,1}:
 7
 3
```

```
 2
 5

julia> B = [7, 6, 42, 6, 7, 42];

julia> sort!(B);  # unique! is able to process sorted data much more efficiently.

julia> unique!(B)
3-element Array{Int64,1}:
  6
  7
 42
```

source

Base.allunique – Function.

```
allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

Examples

```
julia> a = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> allunique([a, a])
false
```

source

Base.reduce – Method.

```
reduce(op, itr; [init])
```

Reduce the given collection `itr` with the given binary operator `op`. If provided, the initial value `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections.

For empty collections, providing `init` will be necessary, except for some special cases (e.g. when `op` is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of `op`.

Reductions for certain commonly-used operators may have special implementations, and should be used instead: `maximum(itr)`, `minimum(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`.

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `-` because it is undefined whether `reduce(-,[1,2,3])` should be evaluated as `(1-2)-3` or `1-(2-3)`. Use `foldl` or `foldr` instead for guaranteed left or right associativity.

Some operations accumulate error. Parallelism will be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, [2; 3; 4])
24

julia> reduce(*, [2; 3; 4]; init=-1)
-24
```

source

`Base.foldl` — Method.

```
foldl(op, itr; [init])
```

Like `reduce`, but with guaranteed left associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

Examples

```
julia> foldl(=>, 1:4)
((1 => 2) => 3) => 4

julia> foldl(=>, 1:4; init=0)
(((0 => 1) => 2) => 3) => 4
```

source

`Base.foldr` — Method.

```
foldr(op, itr; [init])
```

Like reduce, but with guaranteed right associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

Examples

```
julia> foldr(=>, 1:4)
1 => (2 => (3 => 4))

julia> foldr(=>, 1:4; init=0)
1 => (2 => (3 => (4 => 0)))
```

source

Base.maximum – Function.

```
maximum(f, itr)
```

Returns the largest result of calling function `f` on each element of `itr`.

Examples

```
julia> maximum(length, ["Julion", "Julia", "Jule"])
6
```

source

```
maximum(itr)
```

Returns the largest element in a collection.

Examples

```
julia> maximum(-20.5:10)
9.5

julia> maximum([1,2,3])
3
```

source

```
maximum(A::AbstractArray; dims)
```

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum(A, dims=1)
1×2 Array{Int64,2}:
 3  4

julia> maximum(A, dims=2)
2×1 Array{Int64,2}:
 2
 4
```

source

`Base.maximum!` — Function.

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum!([1; 1], A)
2-element Array{Int64,1}:
 2
 4

julia> maximum!([1 1], A)
1×2 Array{Int64,2}:
 3  4
```

source

Base.minimum – Function.

```
minimum(f, itr)
```

Returns the smallest result of calling function `f` on each element of `itr`.

Examples

```
julia> minimum(length, ["Julion", "Julia", "Jule"])
4
```

source

```
minimum(itr)
```

Returns the smallest element in a collection.

Examples

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1
```

source

```
minimum(A::AbstractArray; dims)
```

Compute the minimum value of an array over the given dimensions. See also the `min(a,b)` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min.(a,b)`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> minimum(A, dims=1)
1×2 Array{Int64,2}:
```

```
 1  2


julia> minimum(A, dims=2)
2×1 Array{Int64,2}:
 1
 3
```

source

Base.minimum! – Function.

```
minimum!(r, A)
```

Compute the minimum value of A over the singleton dimensions of r, and write results to r.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4


julia> minimum!([1; 1], A)
2-element Array{Int64,1}:
 1
 3


julia> minimum!([1 1], A)
1×2 Array{Int64,2}:
 1  2
```

source

Base.extrema – Function.

```
extrema(itr) -> Tuple
```

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

Examples

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)
```

source

```
extrema(f, itr) -> Tuple
```

Compute both the minimum and maximum of `f` applied to each element in `itr` and return them as a 2-tuple. Only one pass is made over `itr`.

> Julia 1.2
>
> This method requires Julia 1.2 or later.

Examples

```
julia> extrema(sin, 0:π)
(0.0, 0.9092974268256817)
```

source

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

Examples

```
julia> A = reshape(Vector(1:2:16), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
  9  13
 11  15

julia> extrema(A, dims = (1,2))
1×1×2 Array{Tuple{Int64,Int64},3}:
```

```
[:, :, 1] =
 (1, 7)

[:, :, 2] =
 (9, 15)
```

source

```
extrema(f, A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum of `f` applied to each element in the given dimensions of `A`.

> Julia 1.2
>
> This method requires Julia 1.2 or later.

source

`Base.argmax` — Function.

```
argmax(itr) -> Integer
```

Return the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

Examples

```
julia> argmax([8,0.1,-9,pi])
1

julia> argmax([1,7,7,6])
2

julia> argmax([1,7,7,NaN])
4
```

source

```
argmax(A; dims) -> indices
```

For an array input, return the indices of the maximum elements over the given dimensions. `NaN` is treated as greater than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> argmax(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)

julia> argmax(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
```

source

`Base.argmin` – Function.

```
argmin(itr) -> Integer
```

Return the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

Examples

```
julia> argmin([8,0.1,-9,pi])
3

julia> argmin([7,1,1,6])
2

julia> argmin([7,1,1,NaN])
4
```

source

```
argmin(A; dims) -> indices
```

For an array input, return the indices of the minimum elements over the given dimensions. `NaN` is treated as less than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> argmin(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)

julia> argmin(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
```

source

`Base.findmax` – Function.

```
findmax(itr) -> (x, index)
```

Return the maximum element of the collection `itr` and its index. If there are multiple maximal elements, then the first one will be returned. If any data element is `NaN`, this element is returned. The result is in line with `max`.

The collection must not be empty.

Examples

```
julia> findmax([8,0.1,-9,pi])
(8.0, 1)

julia> findmax([1,7,7,6])
(7, 2)

julia> findmax([1,7,7,NaN])
(NaN, 4)
```

source

```
findmax(A; dims) -> (maxval, index)
```

For an array input, returns the value and index of the maximum over the given dimensions. `NaN` is treated as greater than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> findmax(A, dims=1)
([3.0 4.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(2, 2)])

julia> findmax(A, dims=2)
([2.0; 4.0], CartesianIndex{2}[CartesianIndex(1, 2); CartesianIndex(2, 2)])
```

source

`Base.findmin` – Function.

```
findmin(itr) -> (x, index)
```

Return the minimum element of the collection `itr` and its index. If there are multiple minimal elements, then the first one will be returned. If any data element is `NaN`, this element is returned. The result is in line with `min`.

The collection must not be empty.

Examples

```
julia> findmin([8,0.1,-9,pi])
(-9.0, 3)

julia> findmin([7,1,1,6])
(1, 2)

julia> findmin([7,1,1,NaN])
(NaN, 4)
```

source

```
findmin(A; dims) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given dimensions. `NaN` is treated as less than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> findmin(A, dims=1)
([1.0 2.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(1, 2)])

julia> findmin(A, dims=2)
([1.0; 3.0], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1)])
```

source

`Base.findmax!` – Function.

```
findmax!(rval, rind, A) -> (maxval, index)
```

Find the maximum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as greater than all other values.

source

`Base.findmin!` – Function.

```
findmin!(rval, rind, A) -> (minval, index)
```

Find the minimum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as less than all other values.

source

`Base.sum` – Function.

```
sum(f, itr)
```

Sum the results of calling function `f` on each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> sum(abs2, [2; 3; 4])
29
```

Note the important difference between `sum(A)` and `reduce(+, A)` for arrays with small integer eltype:

```
julia> sum(Int8[100, 28])
128

julia> reduce(+, Int8[100, 28])
-128
```

In the former case, the integers are widened to system word size and therefore the result is 128. In the latter case, no such widening happens and integer overflow results in -128.

source

```
sum(itr)
```

Returns the sum of all elements in a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> sum(1:20)
210
```

source

```
sum(A::AbstractArray; dims)
```

Sum elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum(A, dims=1)
1×2 Array{Int64,2}:
 4  6

julia> sum(A, dims=2)
2×1 Array{Int64,2}:
 3
 7
```

source

Base.sum! – Function.

```
sum!(r, A)
```

Sum elements of A over the singleton dimensions of r, and write results to r.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Array{Int64,1}:
 3
 7

julia> sum!([1 1], A)
1×2 Array{Int64,2}:
 4  6
```

source

Base.prod – Function.

```
prod(f, itr)
```

Returns the product of `f` applied to each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> prod(abs2, [2; 3; 4])
576
```

source

```
prod(itr)
```

Returns the product of all elements of a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> prod(1:20)
2432902008176640000
```

source

```
prod(A::AbstractArray; dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod(A, dims=1)
1×2 Array{Int64,2}:
```

```
 3  8

julia> prod(A, dims=2)
2×1 Array{Int64,2}:
  2
 12
```

source

Base.prod! – Function.

```
prod!(r, A)
```

Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Array{Int64,1}:
  2
 12

julia> prod!([1 1], A)
1×2 Array{Int64,2}:
 3  8
```

source

Base.any – Method.

```
any(itr) -> Bool
```

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> a = [true,false,false,true]
4-element Array{Bool,1}:
 1
 0
 0
 1

julia> any(a)
true

julia> any((println(i); v) for (i, v) in enumerate(a))
1
true

julia> any([missing, true])
true

julia> any([false, missing])
missing
```

source

Base.any – Method.

```
any(p, itr) -> Bool
```

Determine whether predicate p returns true for any elements of itr, returning true as soon as the first item in itr for which p returns true is encountered (short-circuiting).

If the input contains missing values, return missing if all non-missing values are false (or equivalently, if the input contains no true value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
```

```
2
3
4
true

julia> any(i -> i > 0, [1, missing])
true

julia> any(i -> i > 0, [-1, missing])
missing

julia> any(i -> i > 0, [-1, 0])
false
```

source

Base.any! – Function.

```
any!(r, A)
```

Test whether any values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 1  0
 1  0

julia> any!([1; 1], A)
2-element Array{Int64,1}:
 1
 1

julia> any!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

source

Base.all – Method.

```
all(itr) -> Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> a = [true,false,false,true]
4-element Array{Bool,1}:
 1
 0
 0
 1

julia> all(a)
false

julia> all((println(i); v) for (i, v) in enumerate(a))
1
2
false

julia> all([missing, false])
false

julia> all([true, missing])
missing
```

source

`Base.all` – Method.

```
all(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false

julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

source

Base.all! – Function.

```
all!(r, A)
```

Test whether all values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 1  0
 1  0

julia> all!([1; 1], A)
2-element Array{Int64,1}:
 0
 0

julia> all!([1 1], A)
```

```
1×2 Array{Int64,2}:
 1  0
```

source

Base.count – Function.

```
count(p, itr) -> Integer
count(itr) -> Integer
```

Count the number of elements in `itr` for which predicate `p` returns `true`. If `p` is omitted, counts the number of `true` elements in `itr` (which should be a collection of boolean values).

Examples

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3


julia> count([true, false, true, true])
3
```

source

```
count(
    pattern::Union{AbstractString,Regex},
    string::AbstractString;
    overlap::Bool = false,
)
```

Return the number of matches for `pattern` in `string`. This is equivalent to calling `length(findall(pattern, string))` but more efficient.

If `overlap=true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from disjoint character ranges.

source

```
count([f=identity,] A::AbstractArray; dims=:)
```

Count the number of elements in `A` for which `f` returns `true` over the given dimensions.

Julia 1.5

`dims` keyword was added in Julia 1.5.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> count(<=(2), A, dims=1)
1×2 Array{Int64,2}:
 1  1

julia> count(<=(2), A, dims=2)
2×1 Array{Int64,2}:
 2
 0
```

source

Base.any – Method.

```
any(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
2
3
4
true

julia> any(i -> i > 0, [1, missing])
```

```
true

julia> any(i -> i > 0, [-1, missing])
missing

julia> any(i -> i > 0, [-1, 0])
false
```

source

Base.all – Method.

```
all(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false

julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

source

Base.foreach – Function.

```
foreach(f, c...) -> Nothing
```

Call function `f` on each element of iterable `c`. For multiple iterable arguments, `f` is called elementwise. `foreach` should be used instead of `map` when the results of `f` are not needed, for example in `foreach(println, array)`.

Examples

```
julia> a = 1:3:7;

julia> foreach(x -> println(x^2), a)
1
16
49
```

source

Base.map – Function.

```
map(f, c...) -> collection
```

Transform collection `c` by applying `f` to each element. For multiple collection arguments, apply `f` elementwise.

See also: mapslices

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 11
 22
 33
```

source

Base.map! – Function.

```
map!(function, destination, collection...)
```

Like `map`, but stores the result in `destination` rather than a new collection. `destination` must be at least as large as the first collection.

Examples

```
julia> a = zeros(3);

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Array{Float64,1}:
 2.0
 4.0
 6.0
```

source

```
map!(f, values(dict::AbstractDict))
```

Modifies `dict` by transforming each value from `val` to `f(val)`. Note that the type of `dict` cannot be changed: if `f(val)` is not an instance of the value type of `dict` then it will be converted to the value type if possible and otherwise raise an error.

> Julia 1.2
>
> `map!(f, values(dict::AbstractDict))` requires Julia 1.2 or later.

Examples

```
julia> d = Dict(:a => 1, :b => 2)
Dict{Symbol,Int64} with 2 entries:
  :a => 1
  :b => 2

julia> map!(v -> v-1, values(d))
Base.ValueIterator for a Dict{Symbol,Int64} with 2 entries. Values:
  0
  1
```

source

Base.mapreduce – Method.

```
mapreduce(f, op, itrs...; [init])
```

Apply function `f` to each element(s) in `itrs`, and then reduce the result using the binary function `op`. If provided, `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections. In general, it will be necessary to provide `init` to work with empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, map(f, itr); init=init)`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

> Julia 1.2
>
> `mapreduce` with multiple iterators requires Julia 1.2 or later.

Examples

```
julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `itr`. Use `mapfoldl` or `mapfoldr` instead for guaranteed left or right associativity and invocation of `f` for every value.

source

Base.mapfoldl – Method.

```
mapfoldl(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed left associativity, as in `foldl`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

Base.mapfoldr – Method.

```
mapfoldr(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

Base.first – Function.

```
first(coll)
```

Get the first element of an iterable collection. Return the start point of an AbstractRange even if it is empty.

Examples

```
julia> first(2:2:10)
2

julia> first([1; 2; 3; 4])
1
```

source

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

Base.last – Function.

```
last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling lastindex to get the last index. Return the end point of an AbstractRange even if it is empty.

Examples

```
julia> last(1:2:10)
9

julia> last([1; 2; 3; 4])
4
```

source

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

`Base.front` – Function.

```
front(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the last component of `x`.

Examples

```
julia> Base.front((1,2,3))
(1, 2)

julia> Base.front(())
ERROR: ArgumentError: Cannot call front on an empty tuple.
```

source

`Base.tail` – Function.

```
tail(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the first component of `x`.

Examples

```
julia> Base.tail((1,2,3))
(2, 3)

julia> Base.tail(())
ERROR: ArgumentError: Cannot call tail on an empty tuple.
```

source

Base.step – Function.

```
step(r)
```

Get the step size of an AbstractRange object.

Examples

```
julia> step(1:10)
1

julia> step(1:2:10)
2

julia> step(2.5:0.3:10.9)
0.3

julia> step(range(2.5, stop=10.9, length=85))
0.1
```

source

Base.collect – Method.

```
collect(collection)
```

Return an `Array` of all items in a collection or iterator. For dictionaries, returns `Pair{KeyType, ValType}`. If the argument is array-like or is an iterator with the HasShape trait, the result will have the same shape and number of dimensions as the argument.

Examples

```
julia> collect(1:2:13)
7-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11
 13
```

source

Base.collect – Method.

```
collect(element_type, collection)
```

Return an `Array` with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as `collection`.

Examples

```
julia> collect(Float64, 1:2:5)
3-element Array{Float64,1}:
 1.0
 3.0
 5.0
```

source

Base.filter – Function.

```
filter(f, a)
```

Return a copy of collection `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

> Julia 1.4
>
> Support for `a` as a tuple requires at least Julia 1.4.

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

source

```
filter(f, d::AbstractDict)
```

Return a copy of `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64,String} with 2 entries:
  2 => "b"
  1 => "a"

julia> filter(p->isodd(p.first), d)
Dict{Int64,String} with 1 entry:
  1 => "a"
```

source

```
filter(f, itr::SkipMissing{<:AbstractArray})
```

Return a vector similar to the array wrapped by the given `SkipMissing` iterator but with all missing elements and those for which `f` returns `false` removed.

> Julia 1.2
>
> This method requires Julia 1.2 or later.

Examples

```
julia> x = [1 2; missing 4]
2×2 Array{Union{Missing, Int64},2}:
```

```
 1        2
  missing  4

julia> filter(isodd, skipmissing(x))
1-element Array{Int64,1}:
 1
```

source

Base.filter! – Function.

```
filter!(f, a)
```

Update collection `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

Examples

```
julia> filter!(isodd, Vector(1:10))
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

source

```
filter!(f, d::AbstractDict)
```

Update `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Example

```
julia> d = Dict(1=>"a", 2=>"b", 3=>"c")
Dict{Int64,String} with 3 entries:
  2 => "b"
  3 => "c"
  1 => "a"

julia> filter!(p->isodd(p.first), d)
Dict{Int64,String} with 2 entries:
  3 => "c"
  1 => "a"
```

source

Base.replace – Method.

```
replace(A, old_new::Pair...; [count::Integer])
```

Return a copy of collection A where, for each pair old=>new in old_new, all occurrences of old are replaced by new. Equality is determined using isequal. If count is specified, then replace at most count occurrences in total.

The element type of the result is chosen using promotion (see promote_type) based on the element type of A and on the types of the new values in pairs. If count is omitted and the element type of A is a Union, the element type of the result will not include singleton types which are replaced with values of a different type: for example, Union{T,Missing} will become T if missing is replaced.

See also replace!.

Examples

```
julia> replace([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Array{Int64,1}:
 0
 4
 1
 3

julia> replace([1, missing], missing=>0)
2-element Array{Int64,1}:
 1
 0
```

source

Base.replace – Method.

```
replace(new::Function, A; [count::Integer])
```

Return a copy of A where each value x in A is replaced by new(x). If count is specified, then replace at most count values in total (replacements being defined as new(x) !== x).

Examples

```
julia> replace(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Array{Int64,1}:
```

```
 2
 2
 6
 4

julia> replace(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64,Int64} with 2 entries:
  3 => 4
  1 => 3
```

source

Base.replace! – Function.

```
replace!(A, old_new::Pair...; [count::Integer])
```

For each pair `old=>new` in `old_new`, replace all occurrences of `old` in collection `A` by `new`. Equality is determined using isequal. If `count` is specified, then replace at most `count` occurrences in total. See also replace.

Examples

```
julia> replace!([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Array{Int64,1}:
 0
 4
 1
 3

julia> replace!(Set([1, 2, 3]), 1=>0)
Set{Int64} with 3 elements:
  0
  2
  3
```

source

```
replace!(new::Function, A; [count::Integer])
```

Replace each element `x` in collection `A` by `new(x)`. If `count` is specified, then replace at most `count` values in total (replacements being defined as `new(x) !== x`).

Examples

```
julia> replace!(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Array{Int64,1}:
 2
 2
 6
 4

julia> replace!(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64,Int64} with 2 entries:
  3 => 4
  1 => 3

julia> replace!(x->2x, Set([3, 6]))
Set{Int64} with 2 elements:
  6
  12
```

source

46.5 Indexable Collections

Base.getindex – Function.

```
getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

source

Base.setindex! – Function.

```
setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

source

Base.firstindex – Function.

```
firstindex(collection) -> Integer
firstindex(collection, d) -> Integer
```

Return the first index of `collection`. If `d` is given, return the first index of `collection` along dimension `d`.

Examples

```
julia> firstindex([1,2,4])
1


julia> firstindex(rand(3,4,5), 2)
1
```

source

Base.lastindex – Function.

```
lastindex(collection) -> Integer
lastindex(collection, d) -> Integer
```

Return the last index of `collection`. If `d` is given, return the last index of `collection` along dimension `d`.

The syntaxes `A[end]` and `A[end, end]` lower to `A[lastindex(A)]` and `A[lastindex(A, 1), lastindex(A, 2)]`, respectively.

Examples

```
julia> lastindex([1,2,4])
3


julia> lastindex(rand(3,4,5), 2)
4
```

source

Fully implemented by:

- Array
- BitArray
- AbstractArray
- SubArray

Partially implemented by:

- AbstractRange
- UnitRange
- Tuple
- AbstractString
- Dict
- IdDict
- WeakKeyDict
- NamedTuple

46.6 Dictionaries

Dict is the standard dictionary. Its implementation uses hash as the hashing function for the key, and isequal to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

IdDict is a special hash table where the keys are always object identities.

WeakKeyDict is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table. Like `Dict` it uses `hash` for hashing and `isequal` for equality, unlike `Dict` it does not convert keys on insertion.

Dicts can be created by passing pair objects constructed with `=>` to a Dict constructor: `Dict("A"=>1, "B"=>2)`. This call will attempt to infer type information from the keys and values (i.e. this example creates a `Dict{String, Int64}`).

To explicitly specify types use the syntax `Dict{KeyType,ValueType}(...)`. For example, `Dict{String,Int32}("A"=>1, "B"=>2)`.

Dictionaries may also be created with generators. For example, `Dict(i => f(i) for i = 1:10)`.

Given a dictionary `D`, the syntax `D[x]` returns the value of key `x` (if it exists) or throws an error, and `D[x] = y` stores the key-value pair `x => y` in `D` (replacing any existing value for the key `x`). Multiple arguments to `D[...]` are converted to tuples; for example, the syntax `D[x,y]` is equivalent to `D[(x,y)]`, i.e. it refers to the value keyed by the tuple `(x,y)`.

`Base.AbstractDict` — Type.

```
AbstractDict{K, V}
```

Supertype for dictionary-like types with keys of type `K` and values of type `V`. `Dict`, `IdDict` and other types are subtypes of this. An `AbstractDict{K, V}` should be an iterator of `Pair{K, V}`.

source

`Base.Dict` — Type.

```
Dict([itr])
```

`Dict{K,V}()` constructs a hash table with keys of type `K` and values of type `V`. Keys are compared with `isequal` and hashed with `hash`.

Given a single iterable argument, constructs a `Dict` whose key-value pairs are taken from 2-tuples `(key,value)` generated by the argument.

Examples

```
julia> Dict([("A", 1), ("B", 2)])
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Alternatively, a sequence of pair arguments may be passed.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

source

Base.IdDict – Type.

```
IdDict([itr])
```

`IdDict{K,V}()` constructs a hash table using object-id as hash and `===` as equality with keys of type `K` and values of type `V`.

See `Dict` for further help.

source

Base.WeakKeyDict – Type.

```
WeakKeyDict([itr])
```

`WeakKeyDict()` constructs a hash table where the keys are weak references to objects which may be garbage collected even when referenced in a hash table.

See `Dict` for further help. Note, unlike `Dict`, `WeakKeyDict` does not convert keys on insertion.

source

Base.ImmutableDict – Type.

```
ImmutableDict
```

`ImmutableDict` is a dictionary implemented as an immutable linked list, which is optimal for small dictionaries that are constructed over many individual insertions. Note that it is not possible to remove a value, although it can be partially overridden and hidden by inserting a new value with the same key.

```
ImmutableDict(KV::Pair)
```

Create a new entry in the `ImmutableDict` for a `key => value` pair

- use `(key => value) in dict` to see if this particular combination is in the properties set
- use `get(dict, key, default)` to retrieve the most recent value for a particular key

source

Base.haskey – Function.

```
haskey(collection, key) -> Bool
```

Determine whether a collection has a mapping for a given **key**.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> haskey(D, 'a')
true

julia> haskey(D, 'c')
false
```

source

Base.get – Method.

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

Base.get – Function.

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

```
get(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using `do` block syntax

```
get(dict, key) do
    # default value calculated here
    time()
end
```

source

`Base.get!` – Method.

```
get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> get!(d, "a", 5)
1

julia> get!(d, "d", 4)
4

julia> d
```

```
Dict{String,Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

source

Base.get! – Method.

```
get!(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using `do` block syntax:

```
get!(dict, key) do
    # default value calculated here
    time()
end
```

source

Base.getkey – Function.

```
getkey(collection, key, default)
```

Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> getkey(D, 'a', 1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(D, 'd', 'a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

source

Base.delete! – Function.

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, if any, and return the collection.

Examples

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> delete!(d, "b")
Dict{String,Int64} with 1 entry:
  "a" => 1

julia> delete!(d, "b") # d is left unchanged
Dict{String,Int64} with 1 entry:
  "a" => 1
```

source

Base.pop! – Method.

```
pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
```

```
[...]

julia> pop!(d, "e", 4)
4
```

source

Base.keys – Function.

```
keys(iterator)
```

For an iterator or collection that has keys and values (e.g. arrays and dictionaries), return an iterator over the keys.

source

Base.values – Function.

```
values(iterator)
```

For an iterator or collection that has keys and values, return an iterator over the values. This function simply returns its argument by default, since the elements of a general iterator are normally considered its "values".

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> values(d)
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
  2
  1

julia> values([2])
1-element Array{Int64,1}:
 2
```

source

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. When the values are stored internally in a hash table, as is the case for `Dict`, the order in which they are returned may vary. But `keys(a)` and `values(a)` both iterate `a` and return the elements in the same order.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> collect(values(D))
2-element Array{Int64,1}:
 2
 3
```

source

Base.pairs – Function.

```
pairs(IndexLinear(), A)
pairs(IndexCartesian(), A)
pairs(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `i => x`, where `i` is the index for the element and `x = A[i]`. Identical to `pairs(A)`, except that the style of index can be selected. Also similar to `enumerate(A)`, except `i` will be a valid index for `A`, while `enumerate` always counts from 1 regardless of the indices of `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Mutation of the bounds of the underlying array will invalidate this iterator.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];

julia> for (index, value) in pairs(IndexStyle(A), A)
           println("$index $value")
       end
1 a
2 b
3 c
4 d
```

```
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in pairs(IndexStyle(S), S)
           println("$index $value")
       end
CartesianIndex(1, 1) a
CartesianIndex(2, 1) b
CartesianIndex(1, 2) d
CartesianIndex(2, 2) e
```

See also: IndexStyle, axes.

source

```
pairs(collection)
```

Return an iterator over `key => value` pairs for any collection that maps a set of keys to a set of values. This includes arrays, where the keys are the array indices.

source

Base.merge – Function.

```
merge(d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String,Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String,Int64} with 2 entries:
  "bar" => 4711
```

```
  "baz" => 17

julia> merge(a, b)
Dict{String,Float64} with 3 entries:
  "bar" => 4711.0
  "baz" => 17.0
  "foo" => 0.0

julia> merge(b, a)
Dict{String,Float64} with 3 entries:
  "bar" => 42.0
  "baz" => 17.0
  "foo" => 0.0
```

source

```
merge(a::NamedTuple, bs::NamedTuple...)
```

Construct a new named tuple by merging two or more existing ones, in a left-associative manner. Merging proceeds left-to-right, between pairs of named tuples, and so the order of fields present in both the leftmost and rightmost named tuples take the same position as they are found in the leftmost named tuple. However, values are taken from matching fields in the rightmost named tuple that contains that field. Fields present in only the rightmost named tuple of a pair are appended at the end. A fallback is implemented for when only a single named tuple is supplied, with signature `merge(a::NamedTuple)`.

> Julia 1.1
>
> Merging 3 or more `NamedTuple` requires at least Julia 1.1.

Examples

```
julia> merge((a=1, b=2, c=3), (b=4, d=5))
(a = 1, b = 4, c = 3, d = 5)
```

```
julia> merge((a=1, b=2), (b=3, c=(d=1,)), (c=(d=2,),))
(a = 1, b = 3, c = (d = 2,))
```

source

```
merge(a::NamedTuple, iterable)
```

Interpret an iterable of key-value pairs as a named tuple, and perform a merge.

```
julia> merge((a=1, b=2, c=3), [:b=>4, :d=>5])
(a = 1, b = 4, c = 3, d = 5)
```

source

Base.merge! – Method.

```
merge!(d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. See also merge.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> merge!(d1, d2);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 4
```

source

Missing docstring.

Missing docstring for Base.merge!(::Function, ::AbstractDict, ::AbstractDict...). Check Documenter's build log for details.

Base.sizehint! – Function.

```
sizehint!(s, n)
```

Suggest that collection s reserve capacity for at least n elements. This can improve performance.

source

Base.keytype – Function.

```
keytype(T::Type{<:AbstractArray})
keytype(A::AbstractArray)
```

Return the key type of an array. This is equal to the **eltype** of the result of **keys(...)**, and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> keytype([1, 2, 3]) == Int
true


julia> keytype([1 2; 3 4])
CartesianIndex{2}
```

Julia 1.2

For arrays, this function requires at least Julia 1.2.

source

```
keytype(type)
```

Get the key type of an dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> keytype(Dict(Int32(1) => "foo"))
Int32
```

source

`Base.valtype` – Function.

```
valtype(T::Type{<:AbstractArray})
valtype(A::AbstractArray)
```

Return the value type of an array. This is identical to **eltype** and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> valtype(["one", "two", "three"])
String
```

Julia 1.2

For arrays, this function requires at least Julia 1.2.

source

```
valtype(type)
```

Get the value type of an dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> valtype(Dict(Int32(1) => "foo"))
String
```

source

Fully implemented by:

- `IdDict`
- `Dict`
- `WeakKeyDict`

Partially implemented by:

- `BitSet`
- `Set`
- `EnvDict`
- `Array`
- `BitArray`
- `ImmutableDict`
- `Iterators.Pairs`

46.7 Set-Like Collections

Base.AbstractSet – Type.

```
AbstractSet{T}
```

Supertype for set-like types whose elements are of type `T`. `Set`, `BitSet` and other types are subtypes of this.

source

Base.Set – Type.

```
Set([itr])
```

Construct a `Set` of the values generated by the given iterable object, or an empty set. Should be used instead of `BitSet` for sparse integer sets, or for sets of arbitrary objects.

source

Base.BitSet – Type.

```
BitSet([itr])
```

Construct a sorted set of `Int`s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. If the set will be sparse (for example, holding a few very large integers), use `Set` instead.

source

Base.union – Function.

```
union(s, itrs...)
∪(s, itrs...)
```

Construct the union of sets. Maintain order with arrays.

Examples

```
julia> union([1, 2], [3, 4])
4-element Array{Int64,1}:
 1
 2
 3
```

```
 4

julia> union([1, 2], [2, 4])
3-element Array{Int64,1}:
 1
 2
 4

julia> union([4, 2], 1:2)
3-element Array{Int64,1}:
 4
 2
 1

julia> union(Set([1, 2]), 2:3)
Set{Int64} with 3 elements:
  2
  3
  1
```

source

`Base.union!` — Function.

```
union!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the union of passed in sets and overwrite `s` with the result. Maintain order with arrays.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> union!(a, 1:2:8);

julia> a
Set{Int64} with 5 elements:
  7
  4
  3
  5
  1
```

source

Base.intersect – Function.

```
intersect(s, itrs...)
∩(s, itrs...)
```

Construct the intersection of sets. Maintain order with arrays.

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Array{Int64,1}:
 3

julia> intersect([1, 4, 4, 5, 6], [4, 6, 6, 7, 8])
2-element Array{Int64,1}:
 4
 6

julia> intersect(Set([1, 2]), BitSet([2, 3]))
Set{Int64} with 1 element:
  2
```

source

Base.setdiff – Function.

```
setdiff(s, itrs...)
```

Construct the set of elements in `s` but not in any of the iterables in `itrs`. Maintain order with arrays.

Examples

```
julia> setdiff([1,2,3], [3,4,5])
2-element Array{Int64,1}:
 1
 2
```

source

Base.setdiff! – Function.

```
setdiff!(s, itrs...)
```

Remove from set `s` (in-place) each element of each iterable from `itrs`. Maintain order with arrays.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> setdiff!(a, 1:2:6);

julia> a
Set{Int64} with 1 element:
  4
```

source

`Base.symdiff` – Function.

```
symdiff(s, itrs...)
```

Construct the symmetric difference of elements in the passed in sets. When `s` is not an `AbstractSet`, the order is maintained. Note that in this case the multiplicity of elements matters.

Examples

```
julia> symdiff([1,2,3], [3,4,5], [4,5,6])
3-element Array{Int64,1}:
 1
 2
 6

julia> symdiff([1,2,1], [2, 1, 2])
2-element Array{Int64,1}:
 1
 2

julia> symdiff(unique([1,2,1]), unique([2, 1, 2]))
Int64[]
```

source

`Base.symdiff!` – Function.

```
symdiff!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the symmetric difference of the passed in sets, and overwrite `s` with the result. When `s` is an array, the order is maintained. Note that in this case the multiplicity of elements matters.

source

`Base.intersect!` – Function.

```
intersect!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Intersect all passed in sets and overwrite `s` with the result. Maintain order with arrays.

source

`Base.issubset` – Function.

```
issubset(a, b) -> Bool
⊆(a, b) -> Bool
⊇(b, a) -> Bool
```

Determine whether every element of `a` is also in `b`, using `in`.

Examples

```
julia> issubset([1, 2], [1, 2, 3])
true

julia> [1, 2, 3] ⊆ [1, 2]
false

julia> [1, 2, 3] ⊇ [1, 2]
true
```

source

`Base.:⊈` – Function.

```
⊈(a, b) -> Bool
⊉(b, a) -> Bool
```

Negation of `⊆` and `⊇`, i.e. checks that `a` is not a subset of `b`.

Examples

```
julia> (1, 2) ⊈ (2, 3)
true

julia> (1, 2) ⊈ (1, 2, 3)
false
```

source

Base.:⊊ – Function.

```
⊊(a, b) -> Bool
⫋(b, a) -> Bool
```

Determines if a is a subset of, but not equal to, b.

Examples

```
julia> (1, 2) ⊊ (1, 2, 3)
true

julia> (1, 2) ⊊ (1, 2)
false
```

source

Base.issetequal – Function.

```
issetequal(a, b) -> Bool
```

Determine whether a and b have the same elements. Equivalent to a ⊆ b && b ⊆ a but more efficient when possible.

Examples

```
julia> issetequal([1, 2], [1, 2, 3])
false

julia> issetequal([1, 2], [2, 1])
true
```

source

Fully implemented by:

- BitSet
- Set

Partially implemented by:

- Array

46.8 Dequeues

Base.push! – Function.

```
push!(collection, items...) -> collection
```

Insert one or more **items** in **collection**. If **collection** is an ordered container, the items are inserted at the end (in the given order).

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

If **collection** is ordered, use append! to add all the elements of another collection to it. The result of the preceding example is equivalent to **append!([1, 2, 3], [4, 5, 6])**. For **AbstractSet** objects, union! can be used instead.

source

Base.pop! – Function.

```
pop!(collection) -> item
```

Remove an item in **collection** and return it. If **collection** is an ordered container, the last item is returned.

Examples

```
julia> A=[1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> pop!(A)
3

julia> A
2-element Array{Int64,1}:
 1
 2

julia> S = Set([1, 2])
Set{Int64} with 2 elements:
  2
  1

julia> pop!(S)
2

julia> S
Set{Int64} with 1 element:
  1

julia> pop!(Dict(1=>2))
1 => 2
```

source

```
pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);
```

```
julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

source

Base.pushfirst! – Function.

```
pushfirst!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

Examples

```
julia> pushfirst!([1, 2, 3, 4], 5, 6)
6-element Array{Int64,1}:
 5
 6
 1
 2
 3
 4
```

source

Base.popfirst! – Function.

```
popfirst!(collection) -> item
```

Remove the first `item` from `collection`.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Array{Int64,1}:
```

```
 1
 2
 3
 4
 5
 6

julia> popfirst!(A)
1

julia> A
5-element Array{Int64,1}:
 2
 3
 4
 5
 6
```

source

Base.insert! – Function.

```
insert!(a::Vector, index::Integer, item)
```

Insert an `item` into `a` at the given `index`. `index` is the index of `item` in the resulting `a`.

Examples

```
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

source

Base.deleteat! – Function.

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given `i` and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Array{Int64,1}:
 6
 4
 3
 2
 1
```

source

```
deleteat!(a::Vector, inds)
```

Remove the items at the indices given by `inds`, and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

`inds` can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as `a` with `true` indicating entries to delete.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
[...]
```

source

`Base.splice!` – Function.

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
2

julia> A
5-element Array{Int64,1}:
 6
 5
 4
 3
 1

julia> splice!(A, 5, -1)
1

julia> A
5-element Array{Int64,1}:
  6
  5
  4
  3
 -1

julia> splice!(A, 1, [-1, -2, -3])
6

julia> A
7-element Array{Int64,1}:
 -1
 -2
 -3
  5
```

```
 4
 3
-1
```

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

source

```
splice!(a::Vector, indices, [replacement]) -> items
```

Remove items at specified indices, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gaps. If specified, replacement values from an ordered collection will be spliced in place of the removed items; in this case, `indices` must be a `UnitRange`.

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

> Julia 1.5
>
> Prior to Julia 1.5, `indices` must always be a `UnitRange`.

Examples

```
julia> A = [-1, -2, -3, 5, 4, 3, -1]; splice!(A, 4:3, 2)
Int64[]

julia> A
8-element Array{Int64,1}:
 -1
 -2
 -3
  2
  5
  4
  3
 -1
```

source

`Base.resize!` – Function.

```
resize!(a::Vector, n::Integer) -> Vector
```

Resize `a` to contain `n` elements. If `n` is smaller than the current collection length, the first `n` elements will be retained. If `n` is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Array{Int64,1}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8

julia> a[1:6]
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

source

`Base.append!` — Function.

```
append!(collection, collection2) -> collection.
```

For an ordered container `collection`, add the elements of `collection2` to the end of it.

Examples

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> append!([1, 2, 3], [4, 5, 6])
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

Use `push!` to add individual items to `collection` which are not already themselves in another collection. The result of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

source

`Base.prepend!` — Function.

```
prepend!(a::Vector, items) -> collection
```

Insert the elements of `items` to the beginning of `a`.

Examples

```
julia> prepend!([3],[1,2])
3-element Array{Int64,1}:
 1
 2
 3
```

source

Fully implemented by:

- `Vector` (a.k.a. 1-dimensional `Array`)
- `BitVector` (a.k.a. 1-dimensional `BitArray`)

46.9 Utility Collections

`Base.Pair` — Type.

```
Pair(x, y)
x => y
```

Construct a `Pair` object with type `Pair{typeof(x), typeof(y)}`. The elements are stored in the fields `first` and `second`. They can also be accessed via iteration (but a `Pair` is treated as a single "scalar" for broadcasting operations).

See also: `Dict`

Examples

```
julia> p = "foo" => 7
"foo" => 7

julia> typeof(p)
Pair{String,Int64}

julia> p.first
"foo"

julia> for x in p
           println(x)
       end
foo
7
```

source

`Base.Iterators.Pairs` — Type.

```
Iterators.Pairs(values, keys) <: AbstractDict{eltype(keys), eltype(values)}
```

Transforms an indexable container into an Dictionary-view of the same data. Modifying the key-space of the underlying data may invalidate this object.

source

Chapter 47

Mathematics

47.1 Mathematical Operators

`Base.:-` – Method.

```
-(x)
```

Unary minus operator.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Array{Int64,2}:
 -1  -2
 -3  -4
```

source

`Base.:+` – Function.

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

Examples

```
julia> 1 + 20 + 4
25

julia> +(1, 20, 4)
25
```

source

```
dt::Date + t::Time -> DateTime
```

The addition of a `Date` with a `Time` produces a `DateTime`. The hour, minute, second, and millisecond parts of the `Time` are used along with the year, month, and day of the `Date` to create the new `DateTime`. Non-zero microseconds or nanoseconds in the `Time` type will result in an `InexactError` being thrown.

`Base.:-` — Method.

```
-(x, y)
```

Subtraction operator.

Examples

```
julia> 2 - 3
-1

julia> -(2, 4.5)
-2.5
```

source

`Base.:*` — Method.

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

Examples

```
julia> 2 * 7 * 8
112

julia> *(2, 7, 8)
112
```

source

Base.:/ – Function.

```
/(x, y)
```

Right division operator: multiplication of x by the inverse of y on the right. Gives floating-point results for integer arguments.

Examples

```
julia> 1/2
0.5

julia> 4/2
2.0

julia> 4.5/2
2.25
```

source

Base.:\ – Method.

```
\(x, y)
```

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

Examples

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
2.0

julia> A = [4 3; 2 1]; x = [5, 6];

julia> A \ x
2-element Array{Float64,1}:
  6.5
```

```
 -7.0


julia> inv(A) * x
2-element Array{Float64,1}:
  6.5
 -7.0
```

source

Base.:^ – Method.

```
^(x, y)
```

Exponentiation operator. If `x` is a matrix, computes matrix exponentiation.

If `y` is an `Int` literal (e.g. `2` in `x^2` or `-3` in `x^-3`), the Julia code `x^y` is transformed by the compiler to `Base.literal_pow(^, x, Val(y))`, to enable compile-time specialization on the value of the exponent. (As a default fallback we have `Base.literal_pow(^, x, Val(y)) = ^(x,y)`, where usually `^ == Base.^` unless `^` has been defined in the calling namespace.)

```
julia> 3^5
243


julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4


julia> A^3
2×2 Array{Int64,2}:
 37   54
 81  118
```

source

Base.fma – Function.

```
fma(x, y, z)
```

Computes `x*y+z` without rounding the intermediate result `x*y`. On some systems this is significantly more expensive than `x*y+z`. `fma` is used to improve accuracy in certain algorithms. See `muladd`.

source

Base.muladd – Function.

```
muladd(x, y, z)
```

Combined multiply-add: computes `x*y+z`, but allowing the add and multiply to be merged with each other or with surrounding operations for performance. For example, this may be implemented as an `fma` if the hardware supports it efficiently. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See `fma`.

Examples

```
julia> muladd(3, 2, 1)
7


julia> 3 * 2 + 1
7
```

source

Base.inv – Method.

```
inv(x)
```

Return the multiplicative inverse of `x`, such that `x*inv(x)` or `inv(x)*x` yields `one(x)` (the multiplicative identity) up to roundoff errors.

If `x` is a number, this is essentially the same as `one(x)/x`, but for some types `inv(x)` may be slightly more efficient.

Examples

```
julia> inv(2)
0.5


julia> inv(1 + 2im)
0.2 - 0.4im


julia> inv(1 + 2im) * (1 + 2im)
1.0 + 0.0im


julia> inv(2//3)
3//2
```

Julia 1.2

`inv(::Missing)` requires at least Julia 1.2.

source

`Base.div` – Function.

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean division. Computes `x/y`, truncated to an integer.

Examples

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1

julia> 5.0 ÷ 2
2.0
```

source

`Base.fld` – Function.

```
fld(x, y)
```

Largest integer less than or equal to `x/y`. Equivalent to `div(x, y, RoundDown)`.

See also: `div`

Examples

```
julia> fld(7.3,5.5)
1.0
```

source

`Base.cld` – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to `x/y`. Equivalent to `div(x, y, RoundUp)`.

See also: `div`

Examples

```
julia> cld(5.5,2.2)
3.0
```

source

`Base.mod` – Function.

```
mod(x::Integer, r::AbstractUnitRange)
```

Find `y` in the range `r` such that $x \equiv y (mod n)$, where `n = length(r)`, i.e. `y = mod(x - first(r), n) + first(r)`.

See also: `mod1`.

Examples

```
julia> mod(0, Base.OneTo(3))
3


julia> mod(3, 0:2)
0
```

> Julia 1.3
>
> This method requires at least Julia 1.3.

source

```
mod(x, y)
rem(x, y, RoundDown)
```

The reduction of `x` modulo `y`, or equivalently, the remainder of `x` after floored division by `y`, i.e. `x - y*fld(x,y)` if computed without intermediate rounding.

The result will have the same sign as `y`, and magnitude less than `abs(y)` (with some exceptions, see note below).

> Note
>
> When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to `y`, then it may be rounded to `y`.

```
julia> mod(8, 3)
2

julia> mod(9, 3)
0

julia> mod(8.9, 3)
2.9000000000000004

julia> mod(eps(), 3)
2.220446049250313e-16

julia> mod(-eps(), 3)
3.0
```

source

```
rem(x::Integer, T::Type{<:Integer}) -> T
mod(x::Integer, T::Type{<:Integer}) -> T
%(x::Integer, T::Type{<:Integer}) -> T
```

Find `y::T` such that `x` ≡ `y` (mod n), where n is the number of integers representable in `T`, and `y` is an integer in `[typemin(T),typemax(T)]`. If `T` can represent any integer (e.g. `T == BigInt`), then this operation corresponds to a conversion to `T`.

Examples

```
julia> 129 % Int8
-127
```

source

`Base.rem` – Function.

```
rem(x, y)
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as `x`, and smaller in magnitude than `y`. This value is always exact.

Examples

```
julia> x = 15; y = 4;

julia> x % y
3

julia> x == div(x, y) * y + rem(x, y)
true
```

source

Base.Math.rem2pi – Function.

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of x after integer division by 2π, with the quotient rounded according to the rounding mode r. In other words, the quantity

```
x - π2*round(xπ/(2),r)
```

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than `rem(x,2π,r)`

- if `r == RoundNearest`, then the result is in the interval $[-,]$. This will generally be the most accurate result. See also RoundNearest.
- if `r == RoundToZero`, then the result is in the interval $[0, 2]$ if x is positive,. or $[-2, 0]$ otherwise. See also RoundToZero.
- if `r == RoundDown`, then the result is in the interval $[0, 2]$. See also RoundDown.
- if `r == RoundUp`, then the result is in the interval $[-2, 0]$. See also RoundUp.

Examples

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485

julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

source

Base.Math.mod2pi – Function.

```
mod2pi(x)
```

Modulus after division by `2π`, returning in the range $[0, 2)$.

This function computes a floating point representation of the modulus after division by numerically exact `2π`, and is therefore not exactly the same as `mod(x,2π)`, which would compute the modulus of `x` relative to division by the floating-point number `2π`.

> Note
>
> Depending on the format of the input value, the closest representable value to `2π` may be less than `2π`. For example, the expression `mod2pi(2π)` will not return `0`, because the intermediate value of `2*π` is a `Float64` and `2*Float64(π) < 2*big(π)`. See `rem2pi` for more refined control of this behavior.

Examples

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

source

`Base.divrem` – Function.

```
divrem(x, y, r::RoundingMode=RoundToZero)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x,y,r), rem(x,y,r))`. Equivalently, with the default value of `r`, this call is equivalent to `(x÷y, x%y)`.

Examples

```
julia> divrem(3,7)
(0, 3)

julia> divrem(7,3)
(2, 1)
```

source

`Base.fldmod` – Function.

```
fldmod(x, y)
```

The floored quotient and modulus after division. A convenience wrapper for `divrem(x, y, RoundDown)`. Equivalent to `(fld(x,y), mod(x,y))`.

source

Base.fld1 – Function.

```
fld1(x, y)
```

Flooring division, returning a value consistent with `mod1(x,y)`

See also: mod1, fldmod1.

Examples

```
julia> x = 15; y = 4;

julia> fld1(x, y)
4

julia> x == fld(x, y) * y + mod(x, y)
true

julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
true
```

source

Base.mod1 – Function.

```
mod1(x, y)
```

Modulus after flooring division, returning a value `r` such that `mod(r, y) == mod(x, y)` in the range $(0, y]$ for positive `y` and in the range $[y, 0)$ for negative `y`.

See also: fld1, fldmod1.

Examples

```
julia> mod1(4, 2)
2

julia> mod1(4, 3)
1
```

source

Base.fldmod1 – Function.

```
fldmod1(x, y)
```

Return (fld1(x,y), mod1(x,y)).

See also: fld1, mod1.

source

Base.:// – Function.

```
//(num, den)
```

Divide two integers or rational numbers, giving a Rational result.

Examples

```
julia> 3 // 5
3//5

julia> (3 // 5) // (2 // 1)
3//10
```

source

Base.rationalize – Function.

```
rationalize([T<:Integer=Int,] x; tol::Real=eps(x))
```

Approximate floating point number x as a Rational number with components of the given integer type. The result will differ from x by no more than tol.

Examples

```
julia> rationalize(5.6)
28//5

julia> a = rationalize(BigInt, 10.3)
103//10

julia> typeof(numerator(a))
BigInt
```

source

Base.numerator – Function.

```
numerator(x)
```

Numerator of the rational representation of x.

Examples

```
julia> numerator(2//3)
2

julia> numerator(4)
4
```

source

Base.denominator – Function.

```
denominator(x)
```

Denominator of the rational representation of x.

Examples

```
julia> denominator(2//3)
3

julia> denominator(4)
1
```

source

Base.:<< – Function.

```
<<(x, n)
```

Left bit shift operator, x << n. For n >= 0, the result is x shifted left by n bits, filling with 0s. This is equivalent to x * 2^n. For n < 0, this is equivalent to x >> -n.

Examples

```
julia> Int8(3) << 2
12

julia> bitstring(Int8(3))
"00000011"

julia> bitstring(Int8(12))
"00001100"
```

See also >>, >>>.

source

```
<<(B::BitVector, n) -> BitVector
```

Left bit shift operator, `B << n`. For `n >= 0`, the result is `B` with elements shifted `n` positions backwards, filling with `false` values. If `n < 0`, elements are shifted forwards. Equivalent to `B >> -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
 1
 0
 1
 0
 0

julia> B << 1
5-element BitArray{1}:
 0
 1
 0
 0
 0

julia> B << -1
5-element BitArray{1}:
 0
 1
 0
```

```
1
0
```

source

Base.:>> – Function.

```
>>(x, n)
```

Right bit shift operator, `x >> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with `0`s if `x >= 0`, `1`s if `x < 0`, preserving the sign of `x`. This is equivalent to `fld(x, 2^n)`. For `n < 0`, this is equivalent to `x << -n`.

Examples

```
julia> Int8(13) >> 2
3

julia> bitstring(Int8(13))
"00001101"

julia> bitstring(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(-4))
"11111100"
```

See also >>>, <<.

source

```
>>(B::BitVector, n) -> BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is `B` with elements shifted `n` positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
 1
 0
 1
 0
 0

julia> B >> 1
5-element BitArray{1}:
 0
 1
 0
 1
 0

julia> B >> -1
5-element BitArray{1}:
 0
 1
 0
 0
 0
```

source

`Base.:>>>` – Function.

```
>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with `0`s. For `n < 0`, this is equivalent to `x << -n`.

For `Unsigned` integer types, this is equivalent to `>>`. For `Signed` integer types, this is equivalent to `signed(unsigned(x) >> n)`.

Examples

```
julia> Int8(-14) >>> 2
60
```

```
julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(60))
"00111100"
```

BigInts are treated as if having infinite size, so no filling is required and this is equivalent to >>.

See also >>, <<.

source

```
>>>(B::BitVector, n) -> BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See >> for details and examples.

source

Base.:: – Function.

```
(:)(I::CartesianIndex, J::CartesianIndex)
```

Construct CartesianIndices from two `CartesianIndex`.

> Julia 1.1
>
> This method requires at least Julia 1.1.

Examples

```
julia> I = CartesianIndex(2,1);

julia> J = CartesianIndex(3,3);

julia> I:J
2×3 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(2, 3)
 CartesianIndex(3, 1)  CartesianIndex(3, 2)  CartesianIndex(3, 3)
```

source

```
(:)(start, [step], stop)
```

Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1 (a `UnitRange`) , and `a:s:b` is similar but uses a step size of `s` (a `StepRange`).

`:` is also used in indexing to select whole dimensions and for `Symbol` literals, as in e.g. `:hello`.

source

`Base.range` – Function.

```
range(start[, stop]; length, stop, step=1)
```

Given a starting value, construct a range either by length or from `start` to `stop`, optionally with a given step (defaults to 1, a `UnitRange`). One of `length` or `stop` is required. If `length`, `stop`, and `step` are all specified, they must agree.

If `length` and `stop` are provided and `step` is not, the step size will be computed automatically such that there are `length` linearly spaced elements in the range.

If `step` and `stop` are provided and `length` is not, the overall range length will be computed automatically such that the elements are `step` spaced.

Special care is taken to ensure intermediate values are computed rationally. To avoid this induced overhead, see the `LinRange` constructor.

`stop` may be specified as either a positional or keyword argument.

> Julia 1.1
>
> `stop` as a positional argument requires at least Julia 1.1.

Examples

```
julia> range(1, length=100)
1:100

julia> range(1, stop=100)
1:100

julia> range(1, step=5, length=100)
1:5:496

julia> range(1, step=5, stop=100)
1:5:96
```

```
julia> range(1, 10, length=101)
1.0:0.09:10.0

julia> range(1, 100, step=5)
1:5:96
```

source

Base.OneTo – Type.

```
Base.OneTo(n)
```

Define an `AbstractUnitRange` that behaves like `1:n`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

source

Base.StepRangeLen – Type.

```
StepRangeLen{T,R,S}(ref::R, step::S, len, [offset=1]) where {T,R,S}
StepRangeLen(       ref::R, step::S, len, [offset=1]) where {  R,S}
```

A range `r` where `r[i]` produces values of type `T` (in the second form, `T` is deduced automatically), parameterized by a `ref`erence value, a `step`, and the `len`gth. By default `ref` is the starting value `r[1]`, but alternatively you can supply it as the value of `r[offset]` for some other index `1 <= offset <= len`. In conjunction with `TwicePrecision` this can be used to implement ranges that are free of roundoff error.

source

Base.:== – Function.

```
==(x, y)
```

Generic equality operator. Falls back to `===`. Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding. For collections, `==` is generally called recursively on all contents, though other properties (like the shape for arrays) may also be taken into account.

This operator follows IEEE semantics for floating-point numbers: `0.0 == -0.0` and `NaN != NaN`.

The result is of type **Bool**, except when one of the operands is missing, in which case **missing** is returned (three-valued logic). For collections, **missing** is returned if at least one of the operands contains a **missing** value and all non-missing values are equal. Use isequal or === to always get a **Bool** result.

Implementation

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

isequal falls back to **==**, so new methods of **==** will be used by the Dict type to compare keys. If your type will be used as a dictionary key, it should therefore also implement hash.

source

```
==(x)
```

Create a function that compares its argument to x using ==, i.e. a function equivalent to `y -> y == x`.

The returned function is of type **Base.Fix2{typeof(==)}**, which can be used to implement specialized methods.

source

```
==(a::AbstractString, b::AbstractString) -> Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point).

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

source

Base.:!= – Function.

```
!=(x, y)
≠(x,y)
```

Not-equals comparison operator. Always gives the opposite answer as ==.

Implementation

New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

Examples

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

source

```
!=(x)
```

Create a function that compares its argument to `x` using `!=`, i.e. a function equivalent to `y -> y != x`. The returned function is of type `Base.Fix2{typeof(!=)}`, which can be used to implement specialized methods.

> Julia 1.2
>
> This functionality requires at least Julia 1.2.

source

`Base.:!==` – Function.

```
!==(x, y)
≢(x,y)
```

Always gives the opposite answer as `===`.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a ≢ b
true

julia> a ≢ a
false
```

source

`Base.:<` – Function.

```
<(x, y)
```

Less-than comparison operator. Falls back to `isless`. Because of the behavior of floating-point NaN values, this operator implements a partial order.

Implementation

New numeric types with a canonical partial order should implement this function for two arguments of the new type. Types with a canonical total order should implement `isless` instead. (x < y) | (x == y)

Examples

```
julia> 'a' < 'b'
true

julia> "abc" < "abd"
true

julia> 5 < 3
false
```

source

```
<(x)
```

Create a function that compares its argument to `x` using `<`, i.e. a function equivalent to `y -> y < x`. The returned function is of type `Base.Fix2{typeof(<)}`, which can be used to implement specialized methods.

> Julia 1.2
>
> This functionality requires at least Julia 1.2.

source

`Base.:<=` – Function.

```
<=(x, y)
≤(x,y)
```

Less-than-or-equals comparison operator. Falls back to `(x < y) | (x == y)`.

Examples

```
julia> 'a' <= 'b'
true

julia> 7 ≤ 7 ≤ 9
true

julia> "abc" ≤ "abc"
true

julia> 5 <= 3
false
```

source

```
<=(x)
```

Create a function that compares its argument to `x` using `<=`, i.e. a function equivalent to `y -> y <= x`. The returned function is of type `Base.Fix2{typeof(<=)}`, which can be used to implement specialized methods.

> Julia 1.2
>
> This functionality requires at least Julia 1.2.

source

`Base.:>` — Function.

```
>(x, y)
```

Greater-than comparison operator. Falls back to `y < x`.

Implementation

Generally, new types should implement `<` instead of this function, and rely on the fallback definition `>(x, y) = y < x`.

Examples

```
julia> 'a' > 'b'
false

julia> 7 > 3 > 1
true
```

```
julia> "abc" > "abd"
false

julia> 5 > 3
true
```

source

```
>(x)
```

Create a function that compares its argument to `x` using `>`, i.e. a function equivalent to `y -> y > x`. The returned function is of type `Base.Fix2{typeof(>)}`, which can be used to implement specialized methods.

> Julia 1.2
>
> This functionality requires at least Julia 1.2.

source

`Base.:>=` — Function.

```
>=(x, y)
≥(x,y)
```

Greater-than-or-equals comparison operator. Falls back to `y <= x`.

Examples

```
julia> 'a' >= 'b'
false

julia> 7 ≥ 7 ≥ 3
true

julia> "abc" ≥ "abc"
true

julia> 5 >= 3
true
```

source

```
>=(x)
```

Create a function that compares its argument to `x` using `>=`, i.e. a function equivalent to `y -> y >= x`. The returned function is of type `Base.Fix2{typeof(>=)}`, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

`Base.cmp` — Function.

```
cmp(x,y)
```

Return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. Uses the total order implemented by `isless`.

Examples

```
julia> cmp(1, 2)
-1

julia> cmp(2, 1)
1

julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64})
[...]
```

source

```
cmp(<, x, y)
```

Return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. The first argument specifies a less-than comparison function to use.

source

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return `0` if both strings have the same length and the character at each index is the same in both strings. Return `-1` if `a` is a prefix of `b`, or if `a` comes before `b` in alphabetical order. Return `1` if `b` is a prefix of `a`, or if `b` comes before `a` in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

Base.:~ – Function.

```
~(x)
```

Bitwise not.

Examples

```
julia> ~4
-5

julia> ~10
```

```
-11

julia> ~true
false
```

source

Base.:& – Function.

```
x & y
```

Bitwise and. Implements three-valued logic, returning missing if one operand is missing and the other is true. Add parentheses for function application form: (&)(x, y).

Examples

```
julia> 4 & 10
0

julia> 4 & 12
4

julia> true & missing
missing

julia> false & missing
false
```

source

Base.:| – Function.

```
x | y
```

Bitwise or. Implements three-valued logic, returning missing if one operand is missing and the other is false.

Examples

```
julia> 4 | 10
14

julia> 4 | 1
```

```
5

julia> true | missing
true

julia> false | missing
missing
```

source

Base.xor – Function.

```
xor(x, y)
⊻(x, y)
```

Bitwise exclusive or of `x` and `y`. Implements three-valued logic, returning `missing` if one of the arguments is `missing`.

The infix operation `a ⊻ b` is a synonym for `xor(a,b)`, and `⊻` can be typed by tab-completing `\xor` or `\veebar` in the Julia REPL.

Examples

```
julia> xor(true, false)
true

julia> xor(true, true)
false

julia> xor(true, missing)
missing

julia> false ⊻ false
false

julia> [true; true; false] .⊻ [true; false; false]
3-element BitArray{1}:
 0
 1
 0
```

source

`Base.:!` – Function.

```
!(x)
```

Boolean not. Implements three-valued logic, returning `missing` if `x` is `missing`.

Examples

```
julia> !true
false

julia> !false
true

julia> !missing
missing

julia> .![true false true]
1×3 BitArray{2}:
 0  1  0
```

source

```
!f::Function
```

Predicate function negation: when the argument of `!` is a function, it returns a function which computes the boolean negation of `f`.

Examples

```
julia> str = "∀ ε > 0, ∃ δ > 0: ¦x-y¦ < δ ⇒ ¦f(x)-f(y)¦ < ε"
"∀ ε > 0, ∃ δ > 0: ¦x-y¦ < δ ⇒ ¦f(x)-f(y)¦ < ε"

julia> filter(isletter, str)
"εδxyδfxfyε"

julia> filter(!isletter, str)
"∀  > 0, ∃  > 0: ¦-¦ <  ⇒ ¦()-()¦ < "
```

source

`&&` – Keyword.

```
x && y
```

Short-circuiting boolean AND.

source

`||` – Keyword.

```
x || y
```

Short-circuiting boolean OR.

source

47.2 Mathematical Functions

`Base.isapprox` – Function.

```
isapprox(x, y; rtol::Real=atol>0 ? 0 : √eps, atol::Real=0, nans::Bool=false, norm::Function)
```

Inexact equality comparison: `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` is zero and the default `rtol` depends on the types of `x` and `y`. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of `x` or `y`, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significand digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

`x` and `y` may also be arrays of numbers, in which case `norm` defaults to the usual `norm` function in LinearAlgebra, but may be changed by passing a `norm::Function` keyword argument. (For numbers, `norm` is the same thing as `abs`.) When `x` and `y` are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of `x` and `y` are approximately equal component-wise.

The binary operator `≈` is equivalent to `isapprox` with the default arguments, and `x ≉ y` is equivalent to `!isapprox(x,y)`.

Note that `x ≈ 0` (i.e., comparing to zero with the default tolerances) is equivalent to `x == 0` since the default `atol` is `0`. In such cases, you should either supply an appropriate `atol` (or use `norm(x) ≤ atol`) or rearrange your code (e.g. use `x ≈ y` rather than `x - y ≈ 0`). It is not possible to pick a nonzero `atol` automatically because it depends on the overall scaling (the "units") of your problem: for example, in `x - y ≈ 0`, `atol=1e-9` is an absurdly small tolerance if `x` is the radius of the Earth in meters, but an absurdly large tolerance if `x` is the radius of a Hydrogen atom in meters.

Examples

```
julia> 0.1 ≈ (0.1 - 1e-10)
true

julia> isapprox(10, 11; atol = 2)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0])
true

julia> 1e-10 ≈ 0
false

julia> isapprox(1e-10, 0, atol=1e-8)
true
```

source

```
isapprox(x; kwargs...) / ≈(x; kwargs...)
```

Create a function that compares its argument to `x` using `≈`, i.e. a function equivalent to `y -> y ≈ x`.

The keyword arguments supported here are the same as those in the 2-argument `isapprox`.

source

`Base.sin` — Method.

```
sin(x)
```

Compute sine of `x`, where `x` is in radians.

source

`Base.cos` — Method.

```
cos(x)
```

Compute cosine of `x`, where `x` is in radians.

source

`Base.Math.sincos` — Method.

```
sincos(x)
```

Simultaneously compute the sine and cosine of x, where the x is in radians.

source

Base.tan – Method.

```
tan(x)
```

Compute tangent of x, where x is in radians.

source

Base.Math.sind – Function.

```
sind(x)
```

Compute sine of x, where x is in degrees.

source

Base.Math.cosd – Function.

```
cosd(x)
```

Compute cosine of x, where x is in degrees.

source

Base.Math.tand – Function.

```
tand(x)
```

Compute tangent of x, where x is in degrees.

source

Base.Math.sinpi – Function.

```
sinpi(x)
```

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large x.

source

Base.Math.cospi – Function.

```
cospi(x)
```

Compute `cos`(πx) more accurately than `cos(pi*x)`, especially for large `x`.

source

`Base.sinh` – Method.

```
sinh(x)
```

Compute hyperbolic sine of `x`.

source

`Base.cosh` – Method.

```
cosh(x)
```

Compute hyperbolic cosine of `x`.

source

`Base.tanh` – Method.

```
tanh(x)
```

Compute hyperbolic tangent of `x`.

source

`Base.asin` – Method.

```
asin(x)
```

Compute the inverse sine of `x`, where the output is in radians.

source

`Base.acos` – Method.

```
acos(x)
```

Compute the inverse cosine of `x`, where the output is in radians

source

Base.atan – Method.

```
atan(y)
atan(y, x)
```

Compute the inverse tangent of y or y/x, respectively.

For one argument, this is the angle in radians between the positive x-axis and the point (1, y), returning a value in the interval $[-\pi/2, \pi/2]$.

For two arguments, this is the angle in radians between the positive x-axis and the point (x, y), returning a value in the interval $[-\pi, \pi]$. This corresponds to a standard atan2 function.

source

Base.Math.asind – Function.

```
asind(x)
```

Compute the inverse sine of x, where the output is in degrees.

source

Base.Math.acosd – Function.

```
acosd(x)
```

Compute the inverse cosine of x, where the output is in degrees.

source

Base.Math.atand – Function.

```
atand(y)
atand(y,x)
```

Compute the inverse tangent of y or y/x, respectively, where the output is in degrees.

source

Base.Math.sec – Method.

```
sec(x)
```

Compute the secant of x, where x is in radians.

source

Base.Math.csc – Method.

```
csc(x)
```

Compute the cosecant of x, where x is in radians.

source

Base.Math.cot – Method.

```
cot(x)
```

Compute the cotangent of x, where x is in radians.

source

Base.Math.secd – Function.

```
secd(x)
```

Compute the secant of x, where x is in degrees.

source

Base.Math.cscd – Function.

```
cscd(x)
```

Compute the cosecant of x, where x is in degrees.

source

Base.Math.cotd – Function.

```
cotd(x)
```

Compute the cotangent of x, where x is in degrees.

source

Base.Math.asec – Method.

```
asec(x)
```

Compute the inverse secant of x, where the output is in radians.

source

Base.Math.acsc – Method.

```
acsc(x)
```

Compute the inverse cosecant of x, where the output is in radians.

source

Base.Math.acot – Method.

```
acot(x)
```

Compute the inverse cotangent of x, where the output is in radians.

source

Base.Math.asecd – Function.

```
asecd(x)
```

Compute the inverse secant of x, where the output is in degrees.

source

Base.Math.acscd – Function.

```
acscd(x)
```

Compute the inverse cosecant of x, where the output is in degrees.

source

Base.Math.acotd – Function.

```
acotd(x)
```

Compute the inverse cotangent of x, where the output is in degrees.

source

Base.Math.sech – Method.

```
sech(x)
```

Compute the hyperbolic secant of x.

source

Base.Math.csch – Method.

```
csch(x)
```

Compute the hyperbolic cosecant of x.

source

Base.Math.coth – Method.

```
coth(x)
```

Compute the hyperbolic cotangent of x.

source

Base.asinh – Method.

```
asinh(x)
```

Compute the inverse hyperbolic sine of x.

source

Base.acosh – Method.

```
acosh(x)
```

Compute the inverse hyperbolic cosine of x.

source

Base.atanh – Method.

```
atanh(x)
```

Compute the inverse hyperbolic tangent of `x`.

source

Base.Math.asech – Method.

```
asech(x)
```

Compute the inverse hyperbolic secant of `x`.

source

Base.Math.acsch – Method.

```
acsch(x)
```

Compute the inverse hyperbolic cosecant of `x`.

source

Base.Math.acoth – Method.

```
acoth(x)
```

Compute the inverse hyperbolic cotangent of `x`.

source

Base.Math.sinc – Function.

```
sinc(x)
```

Compute $\mathsf{sin}(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

source

Base.Math.cosc – Function.

```
cosc(x)
```

Compute $\mathsf{cos}(\pi x)/x - \mathsf{sin}(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

source

Base.Math.deg2rad – Function.

```
deg2rad(x)
```

Convert x from degrees to radians.

Examples

```
julia> deg2rad(90)
1.5707963267948966
```

source

Base.Math.rad2deg – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

Examples

```
julia> rad2deg(pi)
180.0
```

source

Base.Math.hypot – Function.

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{|x|^2 + |y|^2}$ avoiding overflow and underflow.

This code is an implementation of the algorithm described in: An Improved Algorithm for `hypot(a,b)` by Carlos F. Borges The article is available online at ArXiv at the link https://arxiv.org/abs/1904.09481

Examples

```
julia> a = Int64(10)^10;

julia> hypot(a, a)
1.4142135623730951e10

julia> √(a^2 + a^2) # a^2 overflows
ERROR: DomainError with -2.914184810805068e18:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
```

```
Stacktrace:
[...]

julia> hypot(3, 4im)
5.0
```

source

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum |x_i|^2}$ avoiding overflow and underflow.

Examples

```
julia> hypot(-5.7)
5.7

julia> hypot(3, 4im, 12.0)
13.0
```

source

`Base.log` – Method.

```
log(x)
```

Compute the natural logarithm of `x`. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments to obtain complex results.

Examples

```
julia> log(2)
0.6931471805599453

julia> log(-3)
ERROR: DomainError with -3.0:
log will only return a complex result if called with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

source

`Base.log` – Method.

```
log(b,x)
```

Compute the base `b` logarithm of `x`. Throws `DomainError` for negative `Real` arguments.

Examples

```
julia> log(4,8)
1.5

julia> log(4,2)
0.5

julia> log(-2, 3)
ERROR: DomainError with -2.0:
log will only return a complex result if called with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]

julia> log(2, -3)
ERROR: DomainError with -3.0:
log will only return a complex result if called with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

Note

If `b` is a power of 2 or 10, `log2` or `log10` should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
2.9999999999999996

julia> log10(1000000)/2
3.0
```

source

`Base.log2` – Function.

```
log2(x)
```

Compute the logarithm of x to base 2. Throws DomainError for negative Real arguments.

Examples

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362

julia> log2(-2)
ERROR: DomainError with -2.0:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

source

Base.log10 – Function.

```
log10(x)
```

Compute the logarithm of x to base 10. Throws DomainError for negative Real arguments.

Examples

```
julia> log10(100)
2.0

julia> log10(2)
0.3010299956639812

julia> log10(-2)
ERROR: DomainError with -2.0:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

source

Base.log1p – Function.

```
log1p(x)
```

Accurate natural logarithm of 1+x. Throws DomainError for Real arguments less than -1.

Examples

```
julia> log1p(-0.5)
-0.6931471805599453

julia> log1p(0)
0.0

julia> log1p(-2)
ERROR: DomainError with -2.0:
log1p will only return a complex result if called with a complex argument. Try log1p(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

source

Base.Math.frexp – Function.

```
frexp(val)
```

Return (x,exp) such that x has a magnitude in the interval $[1/2, 1)$ or 0, and val is equal to $x \times 2^{exp}$.

source

Base.exp – Method.

```
exp(x)
```

Compute the natural base exponential of x, in other words e^x.

Examples

```
julia> exp(1.0)
2.718281828459045
```

source

Base.exp2 – Function.

```
exp2(x)
```

Compute the base 2 exponential of x, in other words 2^x.

Examples

```
julia> exp2(5)
32.0
```

source

Base.exp10 – Function.

```
exp10(x)
```

Compute the base 10 exponential of x, in other words 10^x.

Examples

```
julia> exp10(2)
100.0
```

source

```
exp10(x)
```

Compute 10^x.

Examples

```
julia> exp10(2)
100.0

julia> exp10(0.2)
1.5848931924611136
```

source

Base.Math.ldexp – Function.

```
ldexp(x, n)
```

Compute $x \times 2^n$.

Examples

```
julia> ldexp(5., 2)
20.0
```

source

Base.Math.modf – Function.

```
modf(x)
```

Return a tuple `(fpart, ipart)` of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Examples

```
julia> modf(3.5)
(0.5, 3.0)

julia> modf(-3.5)
(-0.5, -3.0)
```

source

Base.expm1 – Function.

```
expm1(x)
```

Accurately compute $e^x - 1$.

source

Base.round – Method.

```
round([T,] x, [r::RoundingMode])
round(x, [r::RoundingMode]; digits::Integer=0, base = 10)
round(x, [r::RoundingMode]; sigdigits::Integer, base = 10)
```

Rounds the number `x`.

Without keyword arguments, `x` is rounded to an integer value, returning a value of type `T`, or of the same type of `x` if no `T` is provided. An `InexactError` will be thrown if the value is not representable by `T`, similar to `convert`.

If the **`digits`** keyword argument is provided, it rounds to the specified number of digits after the decimal place (or before if negative), in base **`base`**.

If the **`sigdigits`** keyword argument is provided, it rounds to the specified number of significant digits, in base **`base`**.

The `RoundingMode` `r` controls the direction of the rounding; the default is `RoundNearest`, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer. Note that **`round`** may give incorrect results if the global rounding mode is changed (see `rounding`).

Examples

```
julia> round(1.7)
2.0

julia> round(Int, 1.7)
2

julia> round(1.5)
2.0

julia> round(2.5)
2.0

julia> round(pi; digits=2)
3.14

julia> round(pi; digits=3, base=2)
3.125

julia> round(123.456; sigdigits=2)
120.0

julia> round(357.913; sigdigits=4, base=2)
352.0
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the `Float64` value represented by `1.15` is actually less than 1.15, yet will be rounded to 1.2.

Chapter 48

Examples

```
julia> x = 1.15
1.15

julia> @sprintf "%.20f" x
"1.14999999999999991118"

julia> x < 115//100
true

julia> round(x, digits=1)
1.2
```

Extensions

To extend `round` to new numeric types, it is typically sufficient to define `Base.round(x::NewType, r::RoundingMode)`.

source

`Base.Rounding.RoundingMode` — Type.

```
RoundingMode
```

A type used for controlling the rounding mode of floating point operations (via `rounding`/`setrounding` functions), or as optional arguments for rounding to the nearest integer (via the `round` function).

Currently supported rounding modes are:

- `RoundNearest` (default)
- `RoundNearestTiesAway`

- RoundNearestTiesUp
- RoundToZero
- RoundFromZero (BigFloat only)
- RoundUp
- RoundDown

source

Base.Rounding.RoundNearest – Constant.

```
RoundNearest
```

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

source

Base.Rounding.RoundNearestTiesAway – Constant.

```
RoundNearestTiesAway
```

Rounds to nearest integer, with ties rounded away from zero (C/C++ round behaviour).

source

Base.Rounding.RoundNearestTiesUp – Constant.

```
RoundNearestTiesUp
```

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript round behaviour).

source

Base.Rounding.RoundToZero – Constant.

```
RoundToZero
```

round using this rounding mode is an alias for trunc.

source

Base.Rounding.RoundFromZero – Constant.

```
RoundFromZero
```

Rounds away from zero. This rounding mode may only be used with `T == BigFloat` inputs to `round`.

Examples

```
julia> BigFloat("1.0000000000000001", 5, RoundFromZero)
1.06
```

source

`Base.Rounding.RoundUp` — Constant.

```
RoundUp
```

`round` using this rounding mode is an alias for `ceil`.

source

`Base.Rounding.RoundDown` — Constant.

```
RoundDown
```

`round` using this rounding mode is an alias for `floor`.

source

`Base.round` — Method.

```
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]])
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; digits=, base=10)
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; sigdigits=, base=10)
```

Return the nearest integral value of the same type as the complex-valued `z` to `z`, breaking ties using the specified `RoundingMode`s. The first `RoundingMode` is used for rounding the real components while the second is used for rounding the imaginary components.

Example

```
julia> round(3.14 + 4.5im)
3.0 + 4.0im
```

source

Base.ceil – Function.

```
ceil([T,] x)
ceil(x; digits::Integer= [, base = 10])
ceil(x; sigdigits::Integer= [, base = 10])
```

`ceil(x)` returns the nearest integral value of the same type as `x` that is greater than or equal to `x`.

`ceil(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for round.

source

Base.floor – Function.

```
floor([T,] x)
floor(x; digits::Integer= [, base = 10])
floor(x; sigdigits::Integer= [, base = 10])
```

`floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`.

`floor(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for round.

source

Base.trunc – Function.

```
trunc([T,] x)
trunc(x; digits::Integer= [, base = 10])
trunc(x; sigdigits::Integer= [, base = 10])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for round.

source

Base.unsafe_trunc – Function.

```
unsafe_trunc(T, x)
```

Return the nearest integral value of type `T` whose absolute value is less than or equal to `x`. If the value is not representable by `T`, an arbitrary value will be returned.

source

`Base.min` – Function.

```
min(x, y, ...)
```

Return the minimum of the arguments. See also the `minimum` function to take the minimum element from a collection.

Examples

```
julia> min(2, 5, 1)
1
```

source

`Base.max` – Function.

```
max(x, y, ...)
```

Return the maximum of the arguments. See also the `maximum` function to take the maximum element from a collection.

Examples

```
julia> max(2, 5, 1)
5
```

source

`Base.minmax` – Function.

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`. See also: `extrema` that returns `(minimum(x), maximum(x))`.

Examples

```
julia> minmax('c','b')
('b', 'c')
```

source

Base.Math.clamp – Function.

```
clamp(x, lo, hi)
```

Return x if lo <= x <= hi. If x > hi, return hi. If x < lo, return lo. Arguments are promoted to a common type.

Examples

```
julia> clamp.([pi, 1.0, big(10.)], 2., 9.)
3-element Array{BigFloat,1}:
 3.14159265358979323846264338327950288419716939937510582097494459230781640628619 8
 2.0
 9.0

julia> clamp.([11,8,5],10,6) # an example where lo > hi
3-element Array{Int64,1}:
  6
  6
 10
```

source

```
clamp(x, T)::T
```

Clamp x between typemin(T) and typemax(T) and convert the result to type T.

Examples

```
julia> clamp(200, Int8)
127
julia> clamp(-200, Int8)
-128
```

source

Base.Math.clamp! – Function.

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in array to the specified range, in-place. See also clamp.

source

Base.abs – Function.

```
abs(x)
```

The absolute value of `x`.

When `abs` is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when `abs` is applied to the minimum representable value of a signed integer. That is, when `x == typemin(typeof(x))`, `abs(x) == x < 0`, not `-x` as might be expected.

Examples

```
julia> abs(-3)
3


julia> abs(1 + im)
1.4142135623730951


julia> abs(typemin(Int64))
-9223372036854775808
```

source

Base.Checked.checked_abs – Function.

```
Base.checked_abs(x)
```

Calculates `abs(x)`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `abs(typemin(Int))`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_neg – Function.

```
Base.checked_neg(x)
```

Calculates `-x`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `-typemin(Int)`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_add – Function.

```
Base.checked_add(x, y)
```

Calculates x+y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_sub – Function.

```
Base.checked_sub(x, y)
```

Calculates x-y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_mul – Function.

```
Base.checked_mul(x, y)
```

Calculates x*y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_div – Function.

```
Base.checked_div(x, y)
```

Calculates div(x,y), checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_rem – Function.

```
Base.checked_rem(x, y)
```

Calculates `x%y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_fld` – Function.

```
Base.checked_fld(x, y)
```

Calculates `fld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_mod` – Function.

```
Base.checked_mod(x, y)
```

Calculates `mod(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_cld` – Function.

```
Base.checked_cld(x, y)
```

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.add_with_overflow` – Function.

```
Base.add_with_overflow(x, y) -> (r, f)
```

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.sub_with_overflow` – Function.

```
Base.sub_with_overflow(x, y) -> (r, f)
```

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.mul_with_overflow` – Function.

```
Base.mul_with_overflow(x, y) -> (r, f)
```

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.abs2` – Function.

```
abs2(x)
```

Squared absolute value of `x`.

Examples

```
julia> abs2(-3)
9
```

source

`Base.copysign` – Function.

```
copysign(x, y) -> z
```

Return `z` which has the magnitude of `x` and the same sign as `y`.

Examples

```
julia> copysign(1, -2)
-1

julia> copysign(-1, 2)
1
```

source

`Base.sign` – Function.

```
sign(x)
```

Return zero if `x==0` and $x/|x|$ otherwise (i.e., ± 1 for real `x`).

source

Base.signbit – Function.

```
signbit(x)
```

Returns `true` if the value of the sign of `x` is negative, otherwise `false`.

Examples

```
julia> signbit(-4)
true

julia> signbit(5)
false

julia> signbit(5.5)
false

julia> signbit(-4.1)
true
```

source

Base.flipsign – Function.

```
flipsign(x, y)
```

Return `x` with its sign flipped if `y` is negative. For example `abs(x) = flipsign(x,x)`.

Examples

```
julia> flipsign(5, 3)
5

julia> flipsign(5, -3)
-5
```

source

Base.sqrt – Method.

```
sqrt(x)
```

Return $\sqrt{x}$. Throws DomainError for negative Real arguments. Use complex negative arguments instead. The prefix operator √ is equivalent to sqrt.

Examples

```
julia> sqrt(big(81))
9.0

julia> sqrt(big(-81))
ERROR: DomainError with -81.0:
NaN result for non-NaN input.
Stacktrace:
 [1] sqrt(::BigFloat) at ./mpfr.jl:501
[...]

julia> sqrt(big(complex(-81)))
0.0 + 9.0im
```

source

Base.isqrt – Function.

```
isqrt(n::Integer)
```

Integer square root: the largest integer m such that m*m <= n.

```
julia> isqrt(5)
2
```

source

Base.Math.cbrt – Function.

```
cbrt(x::Real)
```

Return the cube root of x, i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root when $x < 0$).

The prefix operator ∛ is equivalent to cbrt.

Examples

```
julia> cbrt(big(27))
3.0

julia> cbrt(big(-27))
-3.0
```

source

Base.real – Method.

```
real(z)
```

Return the real part of the complex number z.

Examples

```
julia> real(1 + 3im)
1
```

source

Base.imag – Function.

```
imag(z)
```

Return the imaginary part of the complex number z.

Examples

```
julia> imag(1 + 3im)
3
```

source

Base.reim – Function.

```
reim(z)
```

Return both the real and imaginary parts of the complex number z.

Examples

```
julia> reim(1 + 3im)
(1, 3)
```

source

Base.conj – Function.

```
conj(z)
```

Compute the complex conjugate of a complex number z.

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

source

Base.angle – Function.

```
angle(z)
```

Compute the phase angle in radians of a complex number z.

Examples

```
julia> rad2deg(angle(1 + im))
45.0

julia> rad2deg(angle(1 - im))
-45.0

julia> rad2deg(angle(-1 - im))
-135.0
```

source

Base.cis – Function.

```
cis(z)
```

Return $\exp(iz)$.

Examples

```
julia> cis(π) ≈ -1
true
```

source

Base.binomial – Function.

```
binomial(n::Integer, k::Integer)
```

The binomial coefficient $\binom{n}{k}$, being the coefficient of the kth term in the polynomial expansion of $(1+x)^n$.

If n is non-negative, then it is the number of ways to choose `k` out of `n` items:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ is the `factorial` function.

If n is negative, then it is defined in terms of the identity

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$$

Examples

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10

julia> binomial(-5, 3)
-35
```

See also

- `factorial`

External links

- Binomial coefficient on Wikipedia.

source

Base.factorial – Function.

```
factorial(n::Integer)
```

Factorial of n. If n is an Integer, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if n is not small, but you can use factorial(big(n)) to compute the result exactly in arbitrary precision.

Examples

```
julia> factorial(6)
720

julia> factorial(21)
ERROR: OverflowError: 21 is too large to look up in the table; consider using `factorial(big(21))` instead
Stacktrace:
[...]

julia> factorial(big(21))
51090942171709440000
```

See also

- binomial

External links

- Factorial on Wikipedia.

source

Base.gcd – Function.

```
gcd(x,y)
```

Greatest common (positive) divisor (or zero if x and y are both zero). The arguments may be integer and rational numbers.

> Julia 1.4
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcd(6,9)
3

julia> gcd(6,-9)
3

julia> gcd(6,0)
6

julia> gcd(0,0)
0

julia> gcd(1//3,2//3)
1//3

julia> gcd(1//3,-2//3)
1//3

julia> gcd(1//3,2)
1//3
```

source

Base.lcm – Function.

```
lcm(x,y)
```

Least common (non-negative) multiple. The arguments may be integer and rational numbers.

> Julia 1.4
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> lcm(2,3)
6

julia> lcm(-2,3)
```

```
6

julia> lcm(0,3)
0

julia> lcm(0,0)
0

julia> lcm(1//3,2//3)
2//3

julia> lcm(1//3,-2//3)
2//3

julia> lcm(1//3,2)
2//1
```

source

Base.gcdx – Function.

```
gcdx(x,y)
```

Computes the greatest common (positive) divisor of x and y and their B⊠zout coefficients, i.e. the integer coefficients u and v that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v).

The arguments may be integer and rational numbers.

> Julia 1.4
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcdx(12, 42)
(6, -3, 1)

julia> gcdx(240, 46)
(2, -9, 47)
```

> Note

B⊠zout coefficients are not uniquely defined. `gcdx` returns the minimal B⊠zout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients `u` and `v` are minimal in the sense that $|u| < |y/d|$ and $|v| < |x/d|$. Furthermore, the signs of `u` and `v` are chosen so that `d` is positive. For unsigned integers, the coefficients `u` and `v` might be near their `typemax`, and the identity then holds only via the unsigned integers' modulo arithmetic.

source

`Base.ispow2` – Function.

```
ispow2(n::Integer) -> Bool
```

Test whether `n` is a power of two.

Examples

```
julia> ispow2(4)
true

julia> ispow2(5)
false
```

source

`Base.nextpow` – Function.

```
nextpow(a, x)
```

The smallest `a^n` not less than `x`, where `n` is a non-negative integer. `a` must be greater than 1, and `x` must be greater than 0.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
```

```
25

julia> nextpow(4, 16)
16
```

See also prevpow.

source

Base.prevpow – Function.

```
prevpow(a, x)
```

The largest a^n not greater than x, where n is a non-negative integer. a must be greater than 1, and x must not be less than 1.

Examples

```
julia> prevpow(2, 7)
4

julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

See also nextpow.

source

Base.nextprod – Function.

```
nextprod([k_1, k_2,...], n)
```

Next integer greater than or equal to n that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

Examples

```
julia> nextprod([2, 3], 105)
108

julia> 2^2 * 3^3
108
```

source

Base.invmod – Function.

```
invmod(x,m)
```

Take the inverse of `x` modulo `m`: `y` such that $xy = 1 \pmod m$, with $div(x,y) = 0$. This is undefined for $m = 0$, or if $gcd(x,m) \neq 1$.

Examples

```
julia> invmod(2,5)
3

julia> invmod(2,3)
2

julia> invmod(5,6)
5
```

source

Base.powermod – Function.

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod m$.

Examples

```
julia> powermod(2, 6, 5)
4

julia> mod(2^6, 5)
4
```

```
julia> powermod(5, 2, 20)
5

julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11
```

source

Base.ndigits – Function.

```
ndigits(n::Integer; base::Integer=10, pad::Integer=1)
```

Compute the number of digits in integer `n` written in base `base` (`base` must not be in `[-1, 0, 1]`), optionally padded with zeros to a specified size (the result will never be less than `pad`).

Examples

```
julia> ndigits(12345)
5

julia> ndigits(1022, base=16)
3

julia> string(1022, base=16)
"3fe"

julia> ndigits(123, pad=5)
5
```

source

Base.widemul – Function.

```
widemul(x, y)
```

Multiply `x` and `y`, giving the result as a larger type.

Examples

```
julia> widemul(Float32(3.), 4.)
12.0
```

source

Base.Math.@evalpoly – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k z^{k-1} c[k]$ for the coefficients `c[1]`, `c[2]`, ...; that is, the coefficients are given in ascending order by power of `z`. This macro expands to efficient inline code that uses either Horner's method or, for complex `z`, a more efficient Goertzel-like algorithm.

Examples

```
julia> @evalpoly(3, 1, 0, 1)
10

julia> @evalpoly(2, 1, 0, 1)
5

julia> @evalpoly(2, 1, 1, 1)
7
```

source

Base.FastMath.@fastmath – Macro.

```
@fastmath expr
```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible operation, but results are undefined – be careful when doing this, as it may change numerical results.

This sets the LLVM Fast-Math flags, and corresponds to the `-ffast-math` option in clang. See the notes on performance annotations for more details.

Examples

```
julia> @fastmath 1+2
3
```

```
julia> @fastmath(sin(3))
0.1411200080598672
```

source

Chapter 49

Numbers

49.1 Standard Numeric Types

Abstract number types

`Core.Number` – Type.

```
Number
```

Abstract supertype for all number types.

source

`Core.Real` – Type.

```
Real <: Number
```

Abstract supertype for all real numbers.

source

`Core.AbstractFloat` – Type.

```
AbstractFloat <: Real
```

Abstract supertype for all floating point numbers.

source

`Core.Integer` – Type.

```
Integer <: Real
```

Abstract supertype for all integers.

source

Core.Signed – Type.

```
Signed <: Integer
```

Abstract supertype for all signed integers.

source

Core.Unsigned – Type.

```
Unsigned <: Integer
```

Abstract supertype for all unsigned integers.

source

Base.AbstractIrrational – Type.

```
AbstractIrrational <: Real
```

Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities.

Subtypes `MyIrrational <: AbstractIrrational` should implement at least `==(::MyIrrational, ::MyIrrational)`, `hash(x::MyIrrational, h::UInt)`, and `convert(::Type{F}, x::MyIrrational) where {F <: Union{BigFloat,Float32,Float64}}`.

If a subtype is used to represent values that may occasionally be rational (e.g. a square-root type that represents √n for integers `n` will give a rational result when `n` is a perfect square), then it should also implement `isinteger`, `iszero`, `isone`, and `==` with `Real` values (since all of these default to `false` for `AbstractIrrational` types), as well as defining `hash` to equal that of the corresponding `Rational`.

source

Concrete number types

Core.Float16 – Type.

```
Float16 <: AbstractFloat
```

16-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 5 exponent, 10 fraction bits.

source

Core.Float32 – Type.

```
Float32 <: AbstractFloat
```

32-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 8 exponent, 23 fraction bits.

source

Core.Float64 – Type.

```
Float64 <: AbstractFloat
```

64-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 11 exponent, 52 fraction bits.

source

Base.MPFR.BigFloat – Type.

```
BigFloat <: AbstractFloat
```

Arbitrary precision floating point number type.

source

Core.Bool – Type.

```
Bool <: Integer
```

Boolean type, containing the values `true` and `false`.

`Bool` is a kind of number: `false` is numerically equal to `0` and `true` is numerically equal to `1`. Moreover, `false` acts as a multiplicative "strong zero":

```
julia> false == 0
true
```

```
julia> true == 1
true

julia> 0 * NaN
NaN

julia> false * NaN
0.0
```

source

Core.Int8 – Type.

```
Int8 <: Signed
```

8-bit signed integer type.

source

Core.UInt8 – Type.

```
UInt8 <: Unsigned
```

8-bit unsigned integer type.

source

Core.Int16 – Type.

```
Int16 <: Signed
```

16-bit signed integer type.

source

Core.UInt16 – Type.

```
UInt16 <: Unsigned
```

16-bit unsigned integer type.

source

Core.Int32 – Type.

```
Int32 <: Signed
```

32-bit signed integer type.

source

Core.UInt32 – Type.

```
UInt32 <: Unsigned
```

32-bit unsigned integer type.

source

Core.Int64 – Type.

```
Int64 <: Signed
```

64-bit signed integer type.

source

Core.UInt64 – Type.

```
UInt64 <: Unsigned
```

64-bit unsigned integer type.

source

Core.Int128 – Type.

```
Int128 <: Signed
```

128-bit signed integer type.

source

Core.UInt128 – Type.

```
UInt128 <: Unsigned
```

128-bit unsigned integer type.

source

Base.GMP.BigInt – Type.

```
BigInt <: Signed
```

Arbitrary precision integer type.

source

Base.Complex – Type.

```
Complex{T<:Real} <: Number
```

Complex number type with real and imaginary part of type `T`.

`ComplexF16`, `ComplexF32` and `ComplexF64` are aliases for `Complex{Float16}`, `Complex{Float32}` and `Complex{Float64}` respectively.

source

Base.Rational – Type.

```
Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type `T`. Rationals are checked for overflow.

source

Base.Irrational – Type.

```
Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol `sym`.

source

49.2 Data Formats

Base.digits – Function.

```
digits([T<:Integer], n::Integer; base::T = 10, pad::Integer = 1)
```

Return an array with element type `T` (default `Int`) of the digits of `n` in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indices, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

Examples

```
julia> digits(10, base = 10)
2-element Array{Int64,1}:
 0
 1

julia> digits(10, base = 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits(10, base = 2, pad = 6)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

source

`Base.digits!` – Function.

```
digits!(array, n::Integer; base::Integer = 10)
```

Fills an array of the digits of `n` in the given base. More significant digits are at higher indices. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2,2,2,2], 10, base = 2)
4-element Array{Int64,1}:
```

```
 0
 1
 0
 1

julia> digits!([2,2,2,2,2,2], 10, base = 2)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

source

Base.bitstring – Function.

```
bitstring(n)
```

A string giving the literal bit representation of a number.

Examples

```
julia> bitstring(4)
"0000000000000000000000000000000000000000000000000000000000000100"

julia> bitstring(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

source

Base.parse – Function.

```
parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

Julia 1.1

`parse(Bool, str)` requires at least Julia 1.1.

Examples

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

source

`Base.tryparse` – Function.

```
tryparse(type, str; base)
```

Like `parse`, but returns either a value of the requested type, or `nothing` if the string does not contain a valid number.

source

`Base.big` – Function.

```
big(x)
```

Convert a number to a maximum precision representation (typically `BigInt` or `BigFloat`). See `BigFloat` for information about some pitfalls with floating-point numbers.

source

`Base.signed` – Function.

```
signed(T::Integer)
```

Convert an integer bitstype to the signed type of the same size.

Examples

```
julia> signed(UInt16)
Int16
julia> signed(UInt64)
Int64
```

source

```
signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

source

Base.unsigned – Function.

```
unsigned(T::Integer)
```

Convert an integer bitstype to the unsigned type of the same size.

Examples

```
julia> unsigned(Int16)
UInt16
julia> unsigned(UInt64)
UInt64
```

source

Base.float – Method.

```
float(x)
```

Convert a number or array to a floating point data type.

source

Base.Math.significand – Function.

```
significand(x)
```

Extract the `significand(s)` (a.k.a. mantissa), in binary representation, of a floating-point number. If `x` is a non-zero finite number, then the result will be a number of the same type on the interval $[1, 2)$. Otherwise `x` is returned.

Examples

```
julia> significand(15.2)/15.2
0.125

julia> significand(15.2)*8
15.2
```

source

`Base.Math.exponent` — Function.

```
exponent(x) -> Int
```

Get the exponent of a normalized floating-point number.

source

`Base.complex` — Method.

```
complex(r, [i])
```

Convert real numbers or arrays to complex. `i` defaults to zero.

Examples

```
julia> complex(7)
7 + 0im

julia> complex([1, 2, 3])
3-element Array{Complex{Int64},1}:
 1 + 0im
 2 + 0im
 3 + 0im
```

source

Base.bswap – Function.

```
bswap(n)
```

Reverse the byte order of n.

(See also ntoh and hton to convert between the current native byte order and big-endian order.)

Examples

```
julia> a = bswap(0x10203040)
0x40302010

julia> bswap(a)
0x10203040

julia> string(1, base = 2)
"1"

julia> string(bswap(1), base = 2)
"10000000000000000000000000000000000000000000000000000000000000000"
```

source

Base.hex2bytes – Function.

```
hex2bytes(s::Union{AbstractString,AbstractVector{UInt8}})
```

Given a string or array s of ASCII codes for a sequence of hexadecimal digits, returns a Vector{UInt8} of bytes corresponding to the binary representation: each successive pair of hexadecimal digits in s gives the value of one byte in the return vector.

The length of s must be even, and the returned array has half of the length of s. See also hex2bytes! for an in-place version, and bytes2hex for the inverse.

Examples

```
julia> s = string(12345, base = 16)
"3039"

julia> hex2bytes(s)
2-element Array{UInt8,1}:
```

```
 0x30
 0x39

julia> a = b"01abEF"
6-element Base.CodeUnits{UInt8,String}:
 0x30
 0x31
 0x61
 0x62
 0x45
 0x46

julia> hex2bytes(a)
3-element Array{UInt8,1}:
 0x01
 0xab
 0xef
```

source

Base.hex2bytes! – Function.

```
hex2bytes!(d::AbstractVector{UInt8}, s::Union{String,AbstractVector{UInt8}})
```

Convert an array `s` of bytes representing a hexadecimal string to its binary representation, similar to hex2bytes except that the output is written in-place in `d`. The length of `s` must be exactly twice the length of `d`.

source

Base.bytes2hex – Function.

```
bytes2hex(a::AbstractArray{UInt8}) -> String
bytes2hex(io::IO, a::AbstractArray{UInt8})
```

Convert an array `a` of bytes to its hexadecimal string representation, either returning a `String` via `bytes2hex(a)` or writing the string to an `io` stream via `bytes2hex(io, a)`. The hexadecimal characters are all lowercase.

Examples

```
julia> a = string(12345, base = 16)
"3039"
```

```
julia> b = hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

source

49.3 General Number Functions and Constants

Base.one – Function.

```
one(x)
one(T::type)
```

Return a multiplicative identity for `x`: a value such that `one(x)*x == x*one(x) == x`. Alternatively `one(T)` can take a type `T`, in which case `one` returns a multiplicative identity for any `x` of type `T`.

If possible, `one(x)` returns a value of the same type as `x`, and `one(T)` returns a value of type `T`. However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case, `one(x)` should return an identity value of the same precision (and shape, for matrices) as `x`.

If you want a quantity that is of the same type as `x`, or of type `T`, even if `x` is dimensionful, use `oneunit` instead.

Examples

```
julia> one(3.7)
1.0

julia> one(Int)
1

julia> import Dates; one(Dates.Day(1))
1
```

source

Base.oneunit – Function.

```
oneunit(x::T)
oneunit(T::Type)
```

Returns `T(one(x))`, where `T` is either the type of the argument or (if a type is passed) the argument. This differs from `one` for dimensionful quantities: `one` is dimensionless (a multiplicative identity) while `oneunit` is dimensionful (of the same type as `x`, or of type `T`).

Examples

```
julia> oneunit(3.7)
1.0

julia> import Dates; oneunit(Dates.Day)
1 day
```

source

`Base.zero` – Function.

```
zero(x)
```

Get the additive identity element for the type of `x` (`x` can also specify the type itself).

Examples

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

source

`Base.im` – Constant.

```
im
```

The imaginary unit.

Examples

```
julia> im * im
-1 + 0im
```

source

Base.MathConstants.pi – Constant.

```
π
pi
```

The constant pi.

Examples

```
julia> pi
π = 3.1415926535897...
```

source

Base.MathConstants. – Constant.

```

e
```

The constant ☒.

Examples

```
julia>
  = 2.7182818284590...
```

source

Base.MathConstants.catalan – Constant.

```
catalan
```

Catalan's constant.

Examples

```
julia> Base.MathConstants.catalan
catalan = 0.9159655941772...
```

source

Base.MathConstants.eulergamma – Constant.

```
γ
eulergamma
```

Euler's constant.

Examples

```
julia> Base.MathConstants.eulergamma
γ = 0.5772156649015...
```

source

Base.MathConstants.golden – Constant.

```
ϕ
golden
```

The golden ratio.

Examples

```
julia> Base.MathConstants.golden
ϕ = 1.6180339887498...
```

source

Base.Inf – Constant.

```
Inf, Inf64
```

Positive infinity of type Float64.

source

Base.Inf32 – Constant.

```
Inf32
```

Positive infinity of type Float32.

source

Base.Inf16 – Constant.

```
Inf16
```

Positive infinity of type Float16.

source

Base.NaN – Constant.

```
NaN, NaN64
```

A not-a-number value of type Float64.

source

Base.NaN32 – Constant.

```
NaN32
```

A not-a-number value of type Float32.

source

Base.NaN16 – Constant.

```
NaN16
```

A not-a-number value of type Float16.

source

Base.issubnormal – Function.

```
issubnormal(f) -> Bool
```

Test whether a floating point number is subnormal.

source

Base.isfinite – Function.

```
isfinite(f) -> Bool
```

Test whether a number is finite.

Examples

```
julia> isfinite(5)
true

julia> isfinite(NaN32)
false
```

source

Base.isinf – Function.

```
isinf(f) -> Bool
```

Test whether a number is infinite.

source

Base.isnan – Function.

```
isnan(f) -> Bool
```

Test whether a number value is a NaN, an indeterminate value which is neither an infinity nor a finite number ("not a number").

source

Base.iszero – Function.

```
iszero(x)
```

Return `true` if `x == zero(x)`; if `x` is an array, this checks whether all of the elements of `x` are zero.

Examples

```
julia> iszero(0.0)
true

julia> iszero([1, 9, 0])
```

```
false

julia> iszero([false, 0, 0])
true
```

source

Base.isone – Function.

```
isone(x)
```

Return `true` if `x == one(x)`; if `x` is an array, this checks whether `x` is an identity matrix.

Examples

```
julia> isone(1.0)
true

julia> isone([1 0; 0 2])
false

julia> isone([1 0; 0 true])
true
```

source

Base.nextfloat – Function.

```
nextfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `nextfloat` to `x` if `n >= 0`, or `-n` applications of `prevfloat` if `n < 0`.

source

```
nextfloat(x::AbstractFloat)
```

Return the smallest floating point number `y` of the same type as `x` such `x < y`. If no such `y` exists (e.g. if `x` is `Inf` or `NaN`), then return `x`.

source

Base.prevfloat – Function.

```
prevfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `prevfloat` to `x` if `n >= 0`, or `-n` applications of `nextfloat` if `n < 0`.

source

```
prevfloat(x::AbstractFloat)
```

Return the largest floating point number `y` of the same type as `x` such `y < x`. If no such `y` exists (e.g. if `x` is `-Inf` or `NaN`), then return `x`.

source

Base.isinteger – Function.

```
isinteger(x) -> Bool
```

Test whether `x` is numerically equal to some integer.

Examples

```
julia> isinteger(4.0)
true
```

source

Base.isreal – Function.

```
isreal(x) -> Bool
```

Test whether `x` or all its elements are numerically equal to some real number including infinities and NaNs. `isreal(x)` is true if `isequal(x, real(x))` is true.

Examples

```
julia> isreal(5.)
true

julia> isreal(Inf + 0im)
true

julia> isreal([4.; complex(0,1)])
false
```

source

Core.Float32 — Method.

```
Float32(x [, mode::RoundingMode])
```

Create a `Float32` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
0.3333333f0

julia> Float32(1/3, RoundUp)
0.33333334f0
```

See RoundingMode for available rounding modes.

source

Core.Float64 — Method.

```
Float64(x [, mode::RoundingMode])
```

Create a `Float64` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float64(pi, RoundDown)
3.141592653589793

julia> Float64(pi, RoundUp)
3.1415926535897936
```

See RoundingMode for available rounding modes.

source

Base.Rounding.rounding — Function.

```
rounding(T)
```

Get the current floating point rounding mode for type `T`, controlling the rounding of basic arithmetic functions (+, -, *, / and sqrt) and type conversion.

See RoundingMode for available modes.

source

Base.Rounding.setrounding – Method.

```
setrounding(T, mode)
```

Set the rounding mode of floating point type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default `RoundNearest`.

Note that this is currently only supported for `T == BigFloat`.

> Warning
>
> This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

source

Base.Rounding.setrounding – Method.

```
setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type `T` for the duration of `f`. It is logically equivalent to:

```
old = rounding(T)
setrounding(T, mode)
f()
setrounding(T, old)
```

See `RoundingMode` for available rounding modes.

source

Base.Rounding.get_zero_subnormals – Function.

```
get_zero_subnormals() -> Bool
```

Return `false` if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and `true` if they might be converted to zeros.

> Warning
>
> This function only affects the current thread.

source

Base.Rounding.set_zero_subnormals – Function.

```
set_zero_subnormals(yes::Bool) -> Bool
```

If `yes` is `false`, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns `true` unless `yes==true` but the hardware does not support zeroing of subnormal numbers.

`set_zero_subnormals(true)` can speed up some computations on some hardware. However, it can break identities such as `(x-y==0) == (x==y)`.

> Warning
>
> This function only affects the current thread.

source

Integers

Base.count_ones – Function.

```
count_ones(x::Integer) -> Integer
```

Number of ones in the binary representation of `x`.

Examples

```
julia> count_ones(7)
3
```

source

Base.count_zeros – Function.

```
count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of `x`.

Examples

```
julia> count_zeros(Int32(2 ^ 16 - 1))
16
```

source

Base.leading_zeros – Function.

```
leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of x.

Examples

```
julia> leading_zeros(Int32(1))
31
```

source

Base.leading_ones – Function.

```
leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of x.

Examples

```
julia> leading_ones(UInt32(2 ^ 32 - 2))
31
```

source

Base.trailing_zeros – Function.

```
trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of x.

Examples

```
julia> trailing_zeros(2)
1
```

source

Base.trailing_ones – Function.

```
trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of `x`.

Examples

```
julia> trailing_ones(3)
2
```

source

`Base.isodd` – Function.

```
isodd(x::Integer) -> Bool
```

Return `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

Examples

```
julia> isodd(9)
true

julia> isodd(10)
false
```

source

`Base.iseven` – Function.

```
iseven(x::Integer) -> Bool
```

Return `true` if `x` is even (that is, divisible by 2), and `false` otherwise.

Examples

```
julia> iseven(9)
false

julia> iseven(10)
true
```

source

`Core.@int128_str` – Macro.

```
@int128_str str
@int128_str(str)
```

`@int128_str` parses a string into a Int128 Throws an `ArgumentError` if the string is not a valid integer

source

`Core.@uint128_str` – Macro.

```
@uint128_str str
@uint128_str(str)
```

`@uint128_str` parses a string into a UInt128 Throws an `ArgumentError` if the string is not a valid integer

source

49.4 BigFloats and BigInts

The `BigFloat` and `BigInt` types implements arbitrary-precision floating point and integer arithmetic, respectively. For `BigFloat` the GNU MPFR library is used, and for `BigInt` the GNU Multiple Precision Arithmetic Library (GMP) is used.

`Base.MPFR.BigFloat` – Method.

```
BigFloat(x::Union{Real, AbstractString} [, rounding::RoundingMode=rounding(BigFloat)];
↪  [precision::Integer=precision(BigFloat)])
```

Create an arbitrary precision floating point number from `x`, with precision `precision`. The `rounding` argument specifies the direction in which the result should be rounded if the conversion cannot be done exactly. If not provided, these are set by the current global values.

`BigFloat(x::Real)` is the same as `convert(BigFloat,x)`, except if `x` itself is already `BigFloat`, in which case it will return a value with the precision set to the current global precision; `convert` will always return `x`.

`BigFloat(x::AbstractString)` is identical to `parse`. This is provided for convenience since decimal literals are converted to `Float64` when parsed, so `BigFloat(2.1)` may not yield what you expect.

> Julia 1.1
>
> `precision` as a keyword argument requires at least Julia 1.1. In Julia 1.0 `precision` is the second positional argument (`BigFloat(x, precision)`).

Examples

```
julia> BigFloat(2.1) # 2.1 here is a Float64
2.100000000000000088817841970012523233890533447265625

julia> BigFloat("2.1") # the closest BigFloat to 2.1
2.099999999999999999999999999999999999999999999999999999999999999999999999999986

julia> BigFloat("2.1", RoundUp)
2.100000000000000000000000000000000000000000000000000000000000000000000000000021

julia> BigFloat("2.1", RoundUp, precision=128)
2.100000000000000000000000000000000007
```

See also

- `@big_str`
- `rounding` and `setrounding`
- `precision` and `setprecision`

source

`Base.precision` – Function.

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the significand.

source

`Base.precision` – Method.

```
precision(BigFloat)
```

Get the precision (in bits) currently used for `BigFloat` arithmetic.

source

`Base.MPFR.setprecision` – Function.

```
setprecision([T=BigFloat,] precision::Int)
```

Set the precision (in bits) to be used for `T` arithmetic.

Warning

This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

source

```
setprecision(f::Function, [T=BigFloat,] precision::Integer)
```

Change the `T` arithmetic precision (in bits) for the duration of `f`. It is logically equivalent to:

```
old = precision(BigFloat)
setprecision(BigFloat, precision)
f()
setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

Note: `nextfloat()`, `prevfloat()` do not use the precision mentioned by `setprecision`

source

`Base.GMP.BigInt` – Method.

```
BigInt(x)
```

Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`). The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

Instances can be constructed from strings via `parse`, or using the `big` string literal.

Examples

```
julia> parse(BigInt, "42")
42

julia> big"313"
313

julia> BigInt(10)^19
10000000000000000000
```

source

`Core.@big_str` – Macro.

```
@big_str str
@big_str(str)
```

Parse a string into a `BigInt` or `BigFloat`, and throw an `ArgumentError` if the string is not a valid number. For integers _ is allowed in the string as a separator.

Examples

```
julia> big"123_456"
123456


julia> big"7891.5"
7891.5
```

source

Chapter 50

Strings

Core.AbstractChar — Type.

> The `AbstractChar` type is the supertype of all character implementations in Julia. A character represents a Unicode code point, and can be converted to an integer via the `codepoint` function in order to obtain the numerical value of the code point, or constructed from the same integer. These numerical values determine how characters are compared with `<` and `==`, for example. New `T <: AbstractChar` types should define a `codepoint(::T)` method and a `T(::UInt32)` constructor, at minimum.
>
> A given `AbstractChar` subtype may be capable of representing only a subset of Unicode, in which case conversion from an unsupported `UInt32` value may throw an error. Conversely, the built-in `Char` type represents a superset of Unicode (in order to losslessly encode invalid byte streams), in which case conversion of a non-Unicode value to `UInt32` throws an error. The `isvalid` function can be used to check which codepoints are representable in a given `AbstractChar` type.
>
> Internally, an `AbstractChar` type may use a variety of encodings. Conversion via `codepoint(char)` will not reveal this encoding because it always returns the Unicode value of the character. `print(io, c)` of any `c::AbstractChar` produces an encoding determined by `io` (UTF-8 for all built-in `IO` types), via conversion to `Char` if necessary.
>
> `write(io, c)`, in contrast, may emit an encoding depending on `typeof(c)`, and `read(io, typeof(c))` should read the same encoding as `write`. New `AbstractChar` types must provide their own implementations of `write` and `read`.
>
> source

Core.Char — Type.

```
Char(c::Union{Number,AbstractChar})
```

> `Char` is a 32-bit `AbstractChar` type that is the default representation of characters in Julia. `Char` is the type used for character literals like `'x'` and it is also the element type of `String`.

In order to losslessly represent arbitrary byte streams stored in a `String`, a `Char` value may store information that cannot be converted to a Unicode codepoint — converting such a `Char` to `UInt32` will throw an error. The `isvalid(c::Char)` function can be used to query whether `c` represents a valid Unicode character.

source

`Base.codepoint` – Function.

```
codepoint(c::AbstractChar) -> Integer
```

Return the Unicode codepoint (an unsigned integer) corresponding to the character `c` (or throw an exception if `c` does not represent a valid character). For `Char`, this is a `UInt32` value, but `AbstractChar` types that represent only a subset of Unicode may return a different-sized integer (e.g. `UInt8`).

source

`Base.length` – Method.

```
length(s::AbstractString) -> Int
length(s::AbstractString, i::Integer, j::Integer) -> Int
```

The number of characters in string `s` from indices `i` through `j`. This is computed as the number of code unit indices from `i` to `j` which are valid character indices. With only a single string argument, this computes the number of characters in the entire string. With `i` and `j` arguments it computes the number of indices between `i` and `j` inclusive that are valid indices in the string `s`. In addition to in-bounds values, `i` may take the out-of-bounds value `ncodeunits(s) + 1` and `j` may take the out-of-bounds value `0`.

See also: `isvalid`, `ncodeunits`, `lastindex`, `thisind`, `nextind`, `prevind`

Examples

```
julia> length("jμΛIα")
5
```

source

`Base.sizeof` – Method.

```
sizeof(str::AbstractString)
```

Size, in bytes, of the string `str`. Equal to the number of code units in `str` multiplied by the size, in bytes, of one code unit in `str`.

Examples

```
julia> sizeof("")
0

julia> sizeof("∀")
3
```

source

Base.:* – Method.

```
*(s::Union{AbstractString, AbstractChar}, t::Union{AbstractString, AbstractChar}...) -> AbstractString
```

Concatenate strings and/or characters, producing a String. This is equivalent to calling the string function on the arguments. Concatenation of built-in string types always produces a value of type String but other string types may choose to return a string of a different type as appropriate.

Examples

```
julia> "Hello " * "world"
"Hello world"

julia> 'j' * "ulia"
"julia"
```

source

Base.:^ – Method.

```
^(s::Union{AbstractString,AbstractChar}, n::Integer)
```

Repeat a string or character n times. This can also be written as repeat(s, n).

See also: repeat

Examples

```
julia> "Test "^3
"Test Test Test "
```

source

Base.string – Function.

```
string(n::Integer; base::Integer = 10, pad::Integer = 1)
```

Convert an integer `n` to a string in the given `base`, optionally specifying a number of digits to pad to.

```
julia> string(5, base = 13, pad = 4)
"0005"

julia> string(13, base = 5, pad = 4)
"0023"
```

source

```
string(xs...)
```

Create a string from any values, except `nothing`, using the `print` function.

`string` should usually not be defined directly. Instead, define a method `print(io::IO, x::MyType)`. If `string(x)` for a certain type needs to be highly efficient, then it may make sense to add a method to `string` and define `print(io::IO, x::MyType) = print(io, string(x))` to ensure the functions are consistent.

Examples

```
julia> string("a", 1, true)
"a1true"
```

source

`Base.repeat` – Method.

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also: `^`

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

`Base.repeat` – Method.

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

`Base.repr` – Method.

```
repr(x; context=nothing)
```

Create a string from any value using the `show` function. You should not add methods to `repr`; define a `show` method instead.

The optional keyword argument `context` can be set to an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

Note that `repr(x)` is usually similar to how the value of `x` would be entered in Julia. See also `repr(MIME("text/plain"), x)` to instead return a "pretty-printed" version of `x` designed more for human consumption, equivalent to the REPL display of `x`.

Examples

```
julia> repr(1)
"1"

julia> repr(zeros(3))
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.33333333333333331482961625624739099293947219848632812 5"

julia> repr(big(1/3), context=:compact => true)
"0.333333"
```

source

`Core.String` – Method.

```
String(s::AbstractString)
```

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

source

Base.SubString – Type.

```
SubString(s::AbstractString, i::Integer, j::Integer=lastindex(s))
SubString(s::AbstractString, r::UnitRange{<:Integer})
```

Like getindex, but returns a view into the parent string `s` within range `i:j` or `r` respectively instead of making a copy.

Examples

```
julia> SubString("abc", 1, 2)
"ab"

julia> SubString("abc", 1:2)
"ab"

julia> SubString("abc", 2)
"bc"
```

source

Base.transcode – Function.

```
transcode(T, src)
```

Convert string data between Unicode encodings. `src` is either a `String` or a `Vector{UIntXX}` of UTF-XX code units, where `XX` is 8, 16, or 32. `T` indicates the encoding of the return value: `String` to return a (UTF-8 encoded) `String` or `UIntXX` to return a `Vector{UIntXX}` of UTF-`XX` data. (The alias Cwchar_t can also be used as the integer type, for converting `wchar_t*` strings used by external C libraries.)

The `transcode` function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

source

Base.unsafe_string – Function.

```
unsafe_string(p::Ptr{UInt8}, [length::Integer])
```

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labeled "unsafe" because it will crash if `p` is not a valid memory address to data of the requested length.

source

Base.ncodeunits – Method.

```
ncodeunits(s::AbstractString) -> Int
```

Return the number of code units in a string. Indices that are in bounds to access this string must satisfy `1 ≤ i ≤ ncodeunits(s)`. Not all such indices are valid – they may not be the start of a character, but they will return a code unit value when calling `codeunit(s,i)`.

Examples

```
julia> ncodeunits("The Julia Language")
18

julia> ncodeunits("∫e×")
6

julia> ncodeunits('∫'), ncodeunits('e'), ncodeunits('×')
(3, 1, 2)
```

See also: codeunit, checkbounds, sizeof, length, lastindex

source

Base.codeunit – Function.

```
codeunit(s::AbstractString) -> Type{<:Union{UInt8, UInt16, UInt32}}
```

Return the code unit type of the given string object. For ASCII, Latin-1, or UTF-8 encoded strings, this would be `UInt8`; for UCS-2 and UTF-16 it would be `UInt16`; for UTF-32 it would be `UInt32`. The unit code type need

not be limited to these three types, but it's hard to think of widely used string encodings that don't use one of these units. `codeunit(s)` is the same as `typeof(codeunit(s,1))` when `s` is a non-empty string.

See also: ncodeunits

source

```
codeunit(s::AbstractString, i::Integer) -> Union{UInt8, UInt16, UInt32}
```

Return the code unit value in the string `s` at index `i`. Note that

```
codeunit(s, i) :: codeunit(s)
```

I.e. the value returned by `codeunit(s, i)` is of the type returned by `codeunit(s)`.

Examples

```
julia> a = codeunit("Hello", 2)
0x65

julia> typeof(a)
UInt8
```

See also: ncodeunits, checkbounds

source

Base.codeunits – Function.

```
codeunits(s::AbstractString)
```

Obtain a vector-like object containing the code units of a string. Returns a `CodeUnits` wrapper by default, but `codeunits` may optionally be defined for new string types if necessary.

Examples

```
julia> codeunits("Jυλia")
6-element Base.CodeUnits{UInt8,String}:
 0x4a
 0x75
 0xce
 0xbb
 0x69
 0x61
```

source

Base.ascii – Function.

```
ascii(s::AbstractString)
```

Convert a string to `String` type and check that it contains only ASCII data, otherwise throwing an `ArgumentError` indicating the position of the first non-ASCII byte.

Examples

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
[...]

julia> ascii("abcdefgh")
"abcdefgh"
```

source

Base.@r_str – Macro.

```
@r_str -> Regex
```

Construct a regex, such as `r"^[a-z]*$"`, without interpolation and unescaping (except for quotation mark `"` which still has to be escaped). The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- `i` enables case-insensitive matching
- `m` treats the `^` and `$` tokens as matching the start and end of individual lines, as opposed to the whole string.
- `s` allows the `.` modifier to match newlines.
- `x` enables "comment mode": whitespace is enabled except when escaped with `\`, and `#` is treated as starting a comment.
- `a` disables `UCP` mode (enables ASCII mode). By default `\B`, `\b`, `\D`, `\d`, `\S`, `\s`, `\W`, `\w`, etc. match based on Unicode character properties. With this option, these sequences only match ASCII characters.

See `Regex` if interpolation is needed.

Examples

```
julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

This regex has the first three flags enabled.

source

Base.SubstitutionString – Type.

```
SubstitutionString(substr)
```

Stores the given string `substr` as a `SubstitutionString`, for use in regular expression substitutions. Most commonly constructed using the `@s_str` macro.

```
julia> SubstitutionString("Hello \\g<name>, it's \\1")
s"Hello \\g<name>, it's \\1"

julia> subst = s"Hello \g<name>, it's \1"
s"Hello \\g<name>, it's \\1"

julia> typeof(subst)
SubstitutionString{String}
```

source

Base.@s_str – Macro.

```
@s_str -> SubstitutionString
```

Construct a substitution string, used for regular expression substitutions. Within the string, sequences of the form `\N` refer to the Nth capture group in the regex, and `\g<groupname>` refers to a named capture group with name `groupname`.

```
julia> msg = "#Hello# from Julia";

julia> replace(msg, r"#(.+)# from (?<from>\w+)" => s"FROM: \g<from>; MESSAGE: \1")
"FROM: Julia; MESSAGE: Hello"
```

source

Base.@raw_str – Macro.

```
@raw_str -> String
```

Create a raw string without interpolation and unescaping. The exception is that quotation marks still must be escaped. Backslashes escape both quotation marks and other backslashes, but only when a sequence of backslashes precedes a quote character. Thus, 2n backslashes followed by a quote encodes n backslashes and the end of the literal while 2n+1 backslashes followed by a quote encodes n backslashes followed by a quote character.

Examples

```
julia> println(raw"\ $x")
\ $x

julia> println(raw"\"")
"

julia> println(raw"\\\"")
\"

julia> println(raw"\\x \\\"")
\\x \"
```

source

Base.@b_str – Macro.

```
@b_str
```

Create an immutable byte (`UInt8`) vector using string syntax.

Examples

```
julia> v = b"12\x01\x02"
4-element Base.CodeUnits{UInt8,String}:
 0x31
 0x32
 0x01
 0x02

julia> v[2]
0x32
```

source

Base.Docs.@html_str – Macro.

```
@html_str -> Docs.HTML
```

Create an `HTML` object from a literal string.

source

Base.Docs.@text_str – Macro.

```
@text_str -> Docs.Text
```

Create a `Text` object from a literal string.

source

Base.isvalid – Method.

```
isvalid(value) -> Bool
```

Returns `true` if the given value is valid for its type, which currently can be either `AbstractChar` or `String` or `SubString{String}`.

Examples

```
julia> isvalid(Char(0xd800))
false

julia> isvalid(SubString(String(UInt8[0xfe,0x80,0x80,0x80,0x80,0x80]),1,2))
false

julia> isvalid(Char(0xd799))
true
```

source

Base.isvalid – Method.

```
isvalid(T, value) -> Bool
```

Returns `true` if the given value is valid for that type. Types currently can be either `AbstractChar` or `String`. Values for `AbstractChar` can be of type `AbstractChar` or `UInt32`. Values for `String` can be of that type, or `Vector{UInt8}` or `SubString{String}`.

Examples

```
julia> isvalid(Char, 0xd800)
false

julia> isvalid(String, SubString("thisisvalid",1,5))
true

julia> isvalid(Char, 0xd799)
true
```

source

Base.isvalid – Method.

```
isvalid(s::AbstractString, i::Integer) -> Bool
```

Predicate indicating whether the given index is the start of the encoding of a character in `s` or not. If `isvalid(s, i)` is true then `s[i]` will return the character whose encoding starts at that index, if it's false, then `s[i]` will raise an invalid index error or a bounds error depending on if `i` is in bounds. In order for `isvalid(s, i)` to be an O(1) function, the encoding of `s` must be self-synchronizing this is a basic assumption of Julia's generic string support.

See also: getindex, iterate, thisind, nextind, prevind, length

Examples

```
julia> str = "αβγdef";

julia> isvalid(str, 1)
true

julia> str[1]
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> isvalid(str, 2)
false

julia> str[2]
ERROR: StringIndexError("αβγdef", 2)
Stacktrace:
[...]
```

source

Base.match – Function.

```
match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression `r` in `s` and return a `RegexMatch` object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing `m.match` and the captured sequences can be retrieved by accessing `m.captures` The optional `idx` argument specifies an index at which to start the search.

Examples

```
julia> rx = r"a(.)a"
r"a(.)a"

julia> m = match(rx, "cabac")
RegexMatch("aba", 1="b")

julia> m.captures
1-element Array{Union{Nothing, SubString{String}},1}:
 "b"

julia> m.match
"aba"

julia> match(rx, "cabac", 3) === nothing
true
```

source

Base.eachmatch – Function.

```
eachmatch(r::Regex, s::AbstractString; overlap::Bool=false)
```

Search for all matches of a the regular expression `r` in `s` and return a iterator over the matches. If overlap is `true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

Examples

```
julia> rx = r"a.a"
r"a.a"
```

```
julia> m = eachmatch(rx, "a1a2a3a")
Base.RegexMatchIterator(r"a.a", "a1a2a3a", false)

julia> collect(m)
2-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a3a")

julia> collect(eachmatch(rx, "a1a2a3a", overlap = true))
3-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a2a")
 RegexMatch("a3a")
```

source

Base.isless – Method.

```
isless(a::AbstractString, b::AbstractString) -> Bool
```

Test whether string a comes before string b in alphabetical order (technically, in lexicographical order by Unicode code points).

Examples

```
julia> isless("a", "b")
true

julia> isless("β", "α")
false

julia> isless("a", "a")
false
```

source

Base.:== – Method.

```
==(a::AbstractString, b::AbstractString) -> Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point).

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

source

Base.cmp – Method.

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return `0` if both strings have the same length and the character at each index is the same in both strings. Return `-1` if `a` is a prefix of `b`, or if `a` comes before `b` in alphabetical order. Return `1` if `b` is a prefix of `a`, or if `b` comes before `a` in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

Base.lpad – Function.

```
lpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify `s` and pad the resulting string on the left with `p` to make it `n` characters (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> lpad("March", 10)
"     March"
```

source

Base.rpad – Function.

```
rpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify `s` and pad the resulting string on the right with `p` to make it `n` characters (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> rpad("March", 20)
"March               "
```

source

Base.findfirst – Method.

```
findfirst(pattern::AbstractString, string::AbstractString)
findfirst(pattern::Regex, string::String)
```

Find the first occurrence of `pattern` in `string`. Equivalent to `findnext(pattern, string, firstindex(s))`.

Examples

```
julia> findfirst("z", "Hello to the world") # returns nothing, but not printed in the REPL

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

Base.findnext – Method.

```
findnext(pattern::AbstractString, string::AbstractString, start::Integer)
findnext(pattern::Regex, string::String, start::Integer)
```

Find the next occurrence of `pattern` in `string` starting at position `start`. `pattern` can be either a string, or a regular expression, in which case `string` must be of type `String`.

The return value is a range of indices where the matching sequence is found, such that `s[findnext(x, s, i)] == x`:

`findnext("substring", string, i)` == `start:stop` such that `string[start:stop] == "substring"` and `i <= start`, or `nothing` if unmatched.

Examples

```
julia> findnext("z", "Hello to the world", 1) === nothing
true

julia> findnext("o", "Hello to the world", 6)
8:8

julia> findnext("Lang", "JuliaLang", 2)
6:9
```

source

Base.findnext – Method.

```
findnext(ch::AbstractChar, string::AbstractString, start::Integer)
```

Find the next occurrence of character `ch` in `string` starting at position `start`.

Julia 1.3

This method requires at least Julia 1.3.

Examples

```
julia> findnext('z', "Hello to the world", 1) === nothing
true
```

```
julia> findnext('o', "Hello to the world", 6)
8
```

source

Base.findlast – Method.

```
findlast(pattern::AbstractString, string::AbstractString)
```

Find the last occurrence of `pattern` in `string`. Equivalent to `findprev(pattern, string, lastindex(string))`.

Examples

```
julia> findlast("o", "Hello to the world")
15:15

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

Base.findlast – Method.

```
findlast(ch::AbstractChar, string::AbstractString)
```

Find the last occurrence of character `ch` in `string`.

> Julia 1.3
>
> This method requires at least Julia 1.3.

Examples

```
julia> findlast('p', "happy")
4

julia> findlast('z', "happy") === nothing
true
```

source

Base.findprev – Method.

```
findprev(pattern::AbstractString, string::AbstractString, start::Integer)
```

Find the previous occurrence of `pattern` in `string` starting at position `start`.

The return value is a range of indices where the matching sequence is found, such that `s[findprev(x, s, i)] == x`:

`findprev("substring", string, i)` == `start:stop` such that `string[start:stop] == "substring"` and `stop <= i`, or `nothing` if unmatched.

Examples

```
julia> findprev("z", "Hello to the world", 18) === nothing
true

julia> findprev("o", "Hello to the world", 18)
15:15

julia> findprev("Julia", "JuliaLang", 6)
1:5
```

source

`Base.occursin` – Function.

```
occursin(needle::Union{AbstractString,Regex,AbstractChar}, haystack::AbstractString)
```

Determine whether the first argument is a substring of the second. If `needle` is a regular expression, checks whether `haystack` contains a match.

Examples

```
julia> occursin("Julia", "JuliaLang is pretty cool!")
true

julia> occursin('a', "JuliaLang is pretty cool!")
true

julia> occursin(r"a.a", "aba")
true

julia> occursin(r"a.a", "abba")
false
```

See also: contains.

source

Base.reverse – Method.

```
reverse(s::AbstractString) -> AbstractString
```

Reverses a string. Technically, this function reverses the codepoints in a string and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also reverseind to convert indices in s to indices in reverse(s) and vice-versa, and graphemes from module Unicode to operate on user-visible "characters" (graphemes) rather than codepoints. See also Iterators.reverse for reverse-order iteration without making a copy. Custom string types must implement the reverse function themselves and should typically return a string with the same type and encoding. If they return a string with a different encoding, they must also override reverseind for that string type to satisfy s[reverseind(s,i)] == reverse(s)[i].

Examples

```
julia> reverse("JuliaLang")
"gnaLailuJ"

julia> reverse("axˆe") # combining characters can lead to surprising results
"eˆxa"

julia> using Unicode

julia> join(reverse(collect(graphemes("axˆe")))) # reverses graphemes
"exˆa"
```

source

Base.replace – Method.

```
replace(s::AbstractString, pat=>r; [count::Integer])
```

Search for the given pattern pat in s, and replace each occurrence with r. If count is provided, replace at most count occurrences. pat may be a single character, a vector or a set of characters, a string, or a regular expression. If r is a function, each occurrence is replaced with r(s) where s is the matched substring (when pat is a Regex or AbstractString) or character (when pat is an AbstractChar or a collection of AbstractChar). If pat is a regular expression and r is a SubstitutionString, then capture group references in r are replaced with the corresponding matched text. To remove instances of pat from string, set r to the empty String ("").

Examples

```
julia> replace("Python is a programming language.", "Python" => "Julia")
"Julia is a programming language."

julia> replace("The quick foxes run quickly.", "quick" => "slow", count=1)
"The slow foxes run quickly."

julia> replace("The quick foxes run quickly.", "quick" => "", count=1)
"The  foxes run quickly."

julia> replace("The quick foxes run quickly.", r"fox(es)?" => s"bus\1")
"The quick buses run quickly."
```

source

Base.split – Function.

```
split(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
split(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` into an array of substrings on occurrences of the delimiter(s) `dlm`. `dlm` can be any of the formats allowed by findnext's first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to isspace.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also rsplit.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
```

```
2-element Array{SubString{String},1}:
 "Ma"
 "rch"
```

source

Base.rsplit – Function.

```
rsplit(s::AbstractString; limit::Integer=0, keepempty::Bool=false)
rsplit(s::AbstractString, chars; limit::Integer=0, keepempty::Bool=true)
```

Similar to split, but starting from the end of the string.

Examples

```
julia> a = "M.a.r.c.h"
"M.a.r.c.h"

julia> rsplit(a, ".")
5-element Array{SubString{String},1}:
 "M"
 "a"
 "r"
 "c"
 "h"

julia> rsplit(a, "."; limit=1)
1-element Array{SubString{String},1}:
 "M.a.r.c.h"

julia> rsplit(a, "."; limit=2)
2-element Array{SubString{String},1}:
 "M.a.r.c"
 "h"
```

source

Base.strip – Function.

```
strip([pred=isspace,] str::AbstractString) -> SubString
strip(str::AbstractString, chars) -> SubString
```

Remove leading and trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, vector or set of characters.

> Julia 1.2
>
> The method which accepts a predicate function requires Julia 1.2 or later.

Examples

```
julia> strip("{3, 5}\n", ['{', '}', '\n'])
"3, 5"
```

source

`Base.lstrip` – Function.

```
lstrip([pred=isspace,] str::AbstractString) -> SubString
lstrip(str::AbstractString, chars) -> SubString
```

Remove leading characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

Examples

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

source

`Base.rstrip` – Function.

```
rstrip([pred=isspace,] str::AbstractString) -> SubString
rstrip(str::AbstractString, chars) -> SubString
```

Remove trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

Examples

```
julia> a = rpad("March", 20)
"March               "

julia> rstrip(a)
"March"
```

source

`Base.startswith` – Function.

```
startswith(s::AbstractString, prefix::AbstractString)
```

Return `true` if `s` starts with `prefix`. If `prefix` is a vector or set of characters, test whether the first character of `s` belongs to that set.

See also `endswith`.

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

source

```
startswith(prefix)
```

Create a function that checks whether its argument starts with `prefix`, i.e. a function equivalent to `y -> startswith(y, prefix)`.

The returned function is of type `Base.Fix2{typeof(startswith)}`, which can be used to implement specialized methods.

Julia 1.5

The single argument `startswith(prefix)` requires at least Julia 1.5.

source

```
startswith(s::AbstractString, prefix::Regex)
```

Return `true` if `s` starts with the regex pattern, `prefix`.

Note

`startswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"^...", s)` is faster than `startswith(s, r"...")`.

See also `occursin` and `endswith`.

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> startswith("JuliaLang", r"Julia|Romeo")
true
```

source

`Base.endswith` – Function.

```
endswith(s::AbstractString, suffix::AbstractString)
```

Return `true` if `s` ends with `suffix`. If `suffix` is a vector or set of characters, test whether the last character of `s` belongs to that set.

See also `startswith`.

Examples

```
julia> endswith("Sunday", "day")
true
```

source

```
endswith(suffix)
```

Create a function that checks whether its argument ends with `suffix`, i.e. a function equivalent to `y -> endswith(y, suffix)`.

The returned function is of type `Base.Fix2{typeof(endswith)}`, which can be used to implement specialized methods.

> Julia 1.5
>
> The single argument `endswith(suffix)` requires at least Julia 1.5.

source

```
endswith(s::AbstractString, suffix::Regex)
```

Return `true` if `s` ends with the regex pattern, `suffix`.

> Note
>
> `endswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"...$", s)` is faster than `endswith(s, r"...")`.

See also `occursin` and `startswith`.

> Julia 1.2
>
> This method requires at least Julia 1.2.

Examples

```
julia> endswith("JuliaLang", r"Lang|Roberts")
true
```

source

`Base.first` – Method.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first `n` characters of `s`.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

Base.last – Method.

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

Base.Unicode.uppercase – Function.

```
uppercase(s::AbstractString)
```

Return `s` with all characters converted to uppercase.

Examples

```
julia> uppercase("Julia")
"JULIA"
```

source

Base.Unicode.lowercase – Function.

```
lowercase(s::AbstractString)
```

Return `s` with all characters converted to lowercase.

Examples

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

source

Base.Unicode.titlecase – Function.

```
titlecase(s::AbstractString; [wordsep::Function], strict::Bool=true) -> String
```

Capitalize the first character of each word in `s`; if `strict` is true, every other character is converted to lowercase, otherwise they are left unchanged. By default, all non-letters are considered as word separators; a predicate can be passed as the `wordsep` keyword to determine which characters should be considered as word separators. See also uppercasefirst to capitalize only the first character in `s`.

Examples

```
julia> titlecase("the JULIA programming language")
"The Julia Programming Language"

julia> titlecase("ISS - international space station", strict=false)
"ISS - International Space Station"

julia> titlecase("a-a b-b", wordsep = c->c==' ')
"A-a B-b"
```

source

Base.Unicode.uppercasefirst – Function.

```
uppercasefirst(s::AbstractString) -> String
```

Return `s` with the first character converted to uppercase (technically "title case" for Unicode). See also titlecase to capitalize the first character of every word in `s`.

See also: lowercasefirst, uppercase, lowercase, titlecase

Examples

```
julia> uppercasefirst("python")
"Python"
```

source

Base.Unicode.lowercasefirst – Function.

```
lowercasefirst(s::AbstractString)
```

Return `s` with the first character converted to lowercase.

See also: uppercasefirst, uppercase, lowercase, titlecase

Examples

```
julia> lowercasefirst("Julia")
"julia"
```

source

Base.join – Function.

```
join([io::IO,] strings [, delim [, last]])
```

Join an array of `strings` into a single string, inserting the given delimiter (if any) between adjacent strings. If `last` is given, it will be used instead of `delim` between the last two strings. If `io` is given, the result is written to `io` rather than returned as as a `String`.

`strings` can be any iterable over elements `x` which are convertible to strings via `print(io::IOBuffer, x)`. `strings` will be printed to `io`.

Examples

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"

julia> join([1,2,3,4,5])
"12345"
```

source

Base.chop – Function.

```
chop(s::AbstractString; head::Integer = 0, tail::Integer = 1)
```

Remove the first `head` and the last `tail` characters from `s`. The call `chop(s)` removes the last character from `s`. If it is requested to remove more characters than `length(s)` then an empty string is returned.

Examples

```
julia> a = "March"
"March"

julia> chop(a)
"Marc"

julia> chop(a, head = 1, tail = 2)
"ar"

julia> chop(a, head = 5, tail = 5)
""
```

source

`Base.chomp` – Function.

```
chomp(s::AbstractString) -> SubString
```

Remove a single trailing newline from a string.

Examples

```
julia> chomp("Hello\n")
"Hello"
```

source

`Base.thisind` – Function.

```
thisind(s::AbstractString, i::Integer) -> Int
```

If `i` is in bounds in `s` return the index of the start of the character whose encoding code unit `i` is part of. In other words, if `i` is the start of a character, return `i`; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to 0 or `ncodeunits(s)+1` return `i`. In all other cases throw `BoundsError`.

Examples

```
julia> thisind("α", 0)
0

julia> thisind("α", 1)
1

julia> thisind("α", 2)
1

julia> thisind("α", 3)
3

julia> thisind("α", 4)
ERROR: BoundsError: attempt to access String
  at index [4]
[...]

julia> thisind("α", -1)
ERROR: BoundsError: attempt to access String
  at index [-1]
[...]
```

source

Base.nextind – Function.

```
nextind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts after index `i`. In other words, if `i` is the start of a character, return the start of the next character; if `i` is not the start of a character, move forward until the start of a character and return that index. If `i` is equal to `0` return `1`. If `i` is in bounds but greater or equal to `lastindex(str)` return `ncodeunits(str)+1`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `nextind` for `n==1`. The only difference is that if `n` is so large that applying `nextind` would reach `ncodeunits(str)+1` then each remaining iteration increases the returned value by `1`. This means that in this case `nextind` can return a value greater than `ncodeunits(str)+1`.

- Case `n == 0`

Return `i` only if `i` is a valid index in `s` or is equal to `0`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> nextind("α", 0)
1

julia> nextind("α", 1)
3

julia> nextind("α", 3)
ERROR: BoundsError: attempt to access String
  at index [3]
[...]

julia> nextind("α", 0, 2)
3

julia> nextind("α", 1, 2)
4
```

source

`Base.prevind` – Function.

```
prevind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts before index `i`. In other words, if `i` is the start of a character, return the start of the previous character; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to `1` return `0`. If `i` is equal to `ncodeunits(str)+1` return `lastindex(str)`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `prevind` for `n==1`. The only difference is that if `n` is so large that applying `prevind` would reach `0` then each remaining iteration decreases the returned value by `1`. This means that in this case `prevind` can return a negative value.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `str` or is equal to `ncodeunits(str)+1`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> prevind("α", 3)
1

julia> prevind("α", 1)
0

julia> prevind("α", 0)
ERROR: BoundsError: attempt to access String
  at index [0]
[...]

julia> prevind("α", 2, 2)
0

julia> prevind("α", 2, 3)
-1
```

source

Base.Unicode.textwidth – Function.

```
textwidth(c)
```

Give the number of columns needed to print a character.

Examples

```
julia> textwidth('α')
1

julia> textwidth('　')
2
```

source

```
textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

```
julia> textwidth("March")
5
```

source

Base.isascii – Function.

```
isascii(c::Union{AbstractChar,AbstractString}) -> Bool
```

Test whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

Examples

```
julia> isascii('a')
true

julia> isascii('α')
false

julia> isascii("abc")
true

julia> isascii("αβγ")
false
```

source

Base.Unicode.iscntrl – Function.

```
iscntrl(c::AbstractChar) -> Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

Examples

```
julia> iscntrl('\x01')
true

julia> iscntrl('a')
false
```

source

Base.Unicode.isdigit – Function.

```
isdigit(c::AbstractChar) -> Bool
```

Tests whether a character is a decimal digit (0-9).

Examples

```
julia> isdigit('❤')
false

julia> isdigit('9')
true

julia> isdigit('α')
false
```

source

Base.Unicode.isletter – Function.

```
isletter(c::AbstractChar) -> Bool
```

Test whether a character is a letter. A character is classified as a letter if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

Examples

```
julia> isletter('❤')
false

julia> isletter('α')
true

julia> isletter('9')
false
```

source

Base.Unicode.islowercase – Function.

```
islowercase(c::AbstractChar) -> Bool
```

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

Examples

```
julia> islowercase('α')
true

julia> islowercase('Γ')
false

julia> islowercase('❤')
false
```

source

`Base.Unicode.isnumeric` – Function.

```
isnumeric(c::AbstractChar) -> Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

Note that this broad category includes characters such as ¾ and ☒. Use `isdigit` to check whether a character a decimal digit between 0 and 9.

Examples

```
julia> isnumeric(' ')
true

julia> isnumeric('9')
true

julia> isnumeric('α')
false

julia> isnumeric('❤')
false
```

source

Base.Unicode.isprint – Function.

```
isprint(c::AbstractChar) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

Examples

```
julia> isprint('\x01')
false

julia> isprint('A')
true
```

source

Base.Unicode.ispunct – Function.

```
ispunct(c::AbstractChar) -> Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

Examples

```
julia> ispunct('α')
false

julia> ispunct('/')
true

julia> ispunct(';')
true
```

source

Base.Unicode.isspace – Function.

```
isspace(c::AbstractChar) -> Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters '₩t', '₩n', '₩v', '₩f', '₩r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

Examples

```
julia> isspace('\n')
true

julia> isspace('\r')
true

julia> isspace(' ')
true

julia> isspace('\x20')
true
```

source

Base.Unicode.isuppercase – Function.

```
isuppercase(c::AbstractChar) -> Bool
```

Tests whether a character is an uppercase letter. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

Examples

```
julia> isuppercase('γ')
false

julia> isuppercase('Γ')
true

julia> isuppercase('❤')
false
```

source

Base.Unicode.isxdigit – Function.

```
isxdigit(c::AbstractChar) -> Bool
```

Test whether a character is a valid hexadecimal digit. Note that this does not include `x` (as in the standard `0x` prefix).

Examples

```
julia> isxdigit('a')
true

julia> isxdigit('x')
false
```

source

`Base.escape_string` – Function.

```
escape_string(str::AbstractString[, esc])::AbstractString
escape_string(io, str::AbstractString[, esc::])::Nothing
```

General escaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to `io`.

Backslashes (\) are escaped with a double-backslash ("\\"). Non-printable characters are escaped either with their standard C escape codes, "\0" for NUL (if unambiguous), unicode code point ("\u" prefix) or hex ("\x" prefix).

The optional `esc` argument specifies any additional characters that should also be escaped by a prepending backslash (" is also escaped by default in the first form).

Examples

```
julia> escape_string("aaa\nbbb")
"aaa\\nbbb"

julia> escape_string("\xfe\xff") # invalid utf-8
"\\xfe\\xff"

julia> escape_string(string('\u2135','\0')) # unambiguous
" \\0"

julia> escape_string(string('\u2135','\0','0')) # \0 would be ambiguous
" \\x000"
```

See also

`unescape_string` for the reverse operation.

source

`Base.unescape_string` — Function.

```
unescape_string(str::AbstractString, keep = ())::AbstractString
unescape_string(io, s::AbstractString, keep = ())::Nothing
```

General unescaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to `io`. The argument `keep` specifies a collection of characters which (along with backlashes) are to be kept as they are.

The following escape sequences are recognised:

- Escaped backslash (\\)
- Escaped double-quote (\")
- Standard C escape sequences (\a, \b, \t, \n, \v, \f, \r, \e)
- Unicode BMP code points (\u with 1-4 trailing hex digits)
- All Unicode code points (\U with 1-8 trailing hex digits; max value = 0010ffff)
- Hex bytes (\x with 1-2 trailing hex digits)
- Octal bytes (\ with 1-3 trailing octal digits)

Examples

```
julia> unescape_string("aaa\\nbbb") # C escape sequence
"aaa\nbbb"

julia> unescape_string("\\u03c0") # unicode
"π"

julia> unescape_string("\\101") # octal
"A"

julia> unescape_string("aaa \\g \\n", ['g']) # using `keep` argument
"aaa \\g \n"
```

See also

`escape_string`.

`source`

Chapter 51

Arrays

51.1 Constructors and Types

Core.AbstractArray – Type.

```
AbstractArray{T,N}
```

Supertype for N-dimensional arrays (or array-like types) with elements of type T. Array and other types are subtypes of this. See the manual section on the AbstractArray interface.

source

Base.AbstractVector – Type.

```
AbstractVector{T}
```

Supertype for one-dimensional arrays (or array-like types) with elements of type T. Alias for AbstractArray{T,1}.

source

Base.AbstractMatrix – Type.

```
AbstractMatrix{T}
```

Supertype for two-dimensional arrays (or array-like types) with elements of type T. Alias for AbstractArray{T,2}.

source

Base.AbstractVecOrMat – Constant.

```
AbstractVecOrMat{T}
```

Union type of `AbstractVector{T}` and `AbstractMatrix{T}`.

source

`Core.Array` – Type.

```
Array{T,N} <: AbstractArray{T,N}
```

`N`-dimensional dense array with elements of type `T`.

source

`Core.Array` – Method.

```
Array{T}(undef, dims)
Array{T,N}(undef, dims)
```

Construct an uninitialized `N`-dimensional `Array` containing elements of type `T`. `N` can either be supplied explicitly, as in `Array{T,N}(undef, dims)`, or be determined by the length or number of `dims`. `dims` may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank `N` is supplied explicitly, then it must match the length or number of `dims`. See `undef`.

Examples

```
julia> A = Array{Float64,2}(undef, 2, 3) # N given explicitly
2×3 Array{Float64,2}:
 6.90198e-310  6.90198e-310  6.90198e-310
 6.90198e-310  6.90198e-310  0.0

julia> B = Array{Float64}(undef, 2) # N determined by the input
2-element Array{Float64,1}:
 1.87103e-320
 0.0
```

source

`Core.Array` – Method.

```
Array{T}(nothing, dims)
Array{T,N}(nothing, dims)
```

Construct an `N`-dimensional `Array` containing elements of type `T`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Array{Union{Nothing, String}}(nothing, 2)
2-element Array{Union{Nothing, String},1}:
 nothing
 nothing

julia> Array{Union{Nothing, Int}}(nothing, 2, 3)
2×3 Array{Union{Nothing, Int64},2}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

Core.Array – Method.

```
Array{T}(missing, dims)
Array{T,N}(missing, dims)
```

Construct an `N`-dimensional `Array` containing elements of type `T`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Array{Union{Missing, String}}(missing, 2)
2-element Array{Union{Missing, String},1}:
 missing
 missing

julia> Array{Union{Missing, Int}}(missing, 2, 3)
2×3 Array{Union{Missing, Int64},2}:
 missing  missing  missing
 missing  missing  missing
```

source

Core.UndefInitializer – Type.

```
UndefInitializer
```

Singleton type used in array initialization, indicating the array-constructor-caller would like an uninitialized array. See also `undef`, an alias for `UndefInitializer()`.

Examples

```
julia> Array{Float64,1}(UndefInitializer(), 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

Core.undef – Constant.

```
undef
```

Alias for `UndefInitializer()`, which constructs an instance of the singleton type `UndefInitializer`, used in array initialization to indicate the array-constructor-caller would like an uninitialized array.

Examples

```
julia> Array{Float64,1}(undef, 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

Base.Vector – Type.

```
Vector{T} <: AbstractVector{T}
```

One-dimensional dense array with elements of type `T`, often used to represent a mathematical vector. Alias for `Array{T,1}`.

source

Base.Vector – Method.

```
Vector{T}(undef, n)
```

Construct an uninitialized `Vector{T}` of length `n`. See `undef`.

Examples

```
julia> Vector{Float64}(undef, 3)
3-element Array{Float64,1}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

source

Base.Vector – Method.

```
Vector{T}(nothing, m)
```

Construct a Vector{T} of length m, initialized with nothing entries. Element type T must be able to hold these values, i.e. Nothing <: T.

Examples

```
julia> Vector{Union{Nothing, String}}(nothing, 2)
2-element Array{Union{Nothing, String},1}:
 nothing
 nothing
```

source

Base.Vector – Method.

```
Vector{T}(missing, m)
```

Construct a Vector{T} of length m, initialized with missing entries. Element type T must be able to hold these values, i.e. Missing <: T.

Examples

```
julia> Vector{Union{Missing, String}}(missing, 2)
2-element Array{Union{Missing, String},1}:
 missing
 missing
```

source

Base.Matrix – Type.

```
Matrix{T} <: AbstractMatrix{T}
```

Two-dimensional dense array with elements of type `T`, often used to represent a mathematical matrix. Alias for `Array{T,2}`.

source

`Base.Matrix` – Method.

```
Matrix{T}(undef, m, n)
```

Construct an uninitialized `Matrix{T}` of size `m×n`. See `undef`.

Examples

```
julia> Matrix{Float64}(undef, 2, 3)
2×3 Array{Float64,2}:
 6.93517e-310  6.93517e-310  6.93517e-310
 6.93517e-310  6.93517e-310  1.29396e-320
```

source

`Base.Matrix` – Method.

```
Matrix{T}(nothing, m, n)
```

Construct a `Matrix{T}` of size `m×n`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Matrix{Union{Nothing, String}}(nothing, 2, 3)
2×3 Array{Union{Nothing, String},2}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

`Base.Matrix` – Method.

```
Matrix{T}(missing, m, n)
```

Construct a `Matrix{T}` of size `m×n`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Matrix{Union{Missing, String}}(missing, 2, 3)
2×3 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

source

`Base.VecOrMat` — Constant.

```
VecOrMat{T}
```

Union type of `Vector{T}` and `Matrix{T}`.

source

`Core.DenseArray` — Type.

```
DenseArray{T, N} <: AbstractArray{T,N}
```

`N`-dimensional dense array with elements of type `T`. The elements of a dense array are stored contiguously in memory.

source

`Base.DenseVector` — Type.

```
DenseVector{T}
```

One-dimensional `DenseArray` with elements of type `T`. Alias for `DenseArray{T,1}`.

source

`Base.DenseMatrix` — Type.

```
DenseMatrix{T}
```

Two-dimensional `DenseArray` with elements of type `T`. Alias for `DenseArray{T,2}`.

source

Base.DenseVecOrMat — Constant.

```
DenseVecOrMat{T}
```

Union type of DenseVector{T} and DenseMatrix{T}.

source

Base.StridedArray — Constant.

```
StridedArray{T, N}
```

A hard-coded Union of common array types that follow the strided array interface, with elements of type T and N dimensions.

If A is a StridedArray, then its elements are stored in memory with offsets, which may vary between dimensions but are constant within a dimension. For example, A could have stride 2 in dimension 1, and stride 3 in dimension 2. Incrementing A along dimension d jumps in memory by [strides(A, d)] slots. Strided arrays are particularly important and useful because they can sometimes be passed directly as pointers to foreign language libraries like BLAS.

source

Base.StridedVector — Constant.

```
StridedVector{T}
```

One dimensional StridedArray with elements of type T.

source

Base.StridedMatrix — Constant.

```
StridedMatrix{T}
```

Two dimensional StridedArray with elements of type T.

source

Base.StridedVecOrMat — Constant.

```
StridedVecOrMat{T}
```

Union type of `StridedVector` and `StridedMatrix` with elements of type `T`.

source

`Base.getindex` – Method.

```
getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax `Type[]`. Element values can be specified using `Type[a,b,c,...]`.

Examples

```
julia> Int8[1, 2, 3]
3-element Array{Int8,1}:
 1
 2
 3


julia> getindex(Int8, 1, 2, 3)
3-element Array{Int8,1}:
 1
 2
 3
```

source

`Base.zeros` – Function.

```
zeros([T=Float64,] dims::Tuple)
zeros([T=Float64,] dims...)
```

Create an `Array`, with element type `T`, of all zeros with size specified by `dims`. See also `fill`, `ones`.

Examples

```
julia> zeros(1)
1-element Array{Float64,1}:
 0.0


julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0
```

source

Base.ones – Function.

```
ones([T=Float64,] dims::Tuple)
ones([T=Float64,] dims...)
```

Create an `Array`, with element type `T`, of all ones with size specified by `dims`. See also: `fill`, `zeros`.

Examples

```
julia> ones(1,2)
1×2 Array{Float64,2}:
 1.0  1.0

julia> ones(ComplexF64, 2, 3)
2×3 Array{Complex{Float64},2}:
 1.0+0.0im  1.0+0.0im  1.0+0.0im
 1.0+0.0im  1.0+0.0im  1.0+0.0im
```

source

Base.BitArray – Type.

```
BitArray{N} <: AbstractArray{Bool, N}
```

Space-efficient `N`-dimensional boolean array, using just one bit for each boolean value.

`BitArray`s pack up to 64 values into every 8 bytes, resulting in an 8x space efficiency over `Array{Bool, N}` and allowing some operations to work on 64 values at once.

By default, Julia returns `BitArrays` from broadcasting operations that generate boolean elements (including dotted-comparisons like `.==`) as well as from the functions `trues` and `falses`.

> Note
>
> Due to its packed storage format, concurrent access to the elements of a `BitArray` where at least one of them is a write is not thread safe.

source

Base.BitArray – Method.

```
BitArray(undef, dims::Integer...)
BitArray{N}(undef, dims::NTuple{N,Int})
```

Construct an undef `BitArray` with the given dimensions. Behaves identically to the `Array` constructor. See `undef`.

Examples

```
julia> BitArray(undef, 2, 2)
2×2 BitArray{2}:
 0  0
 0  0

julia> BitArray(undef, (3, 1))
3×1 BitArray{2}:
 0
 0
 0
```

source

`Base.BitArray` – Method.

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

```
julia> BitArray([1 0; 0 1])
2×2 BitArray{2}:
 1  0
 0  1

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitArray{2}:
 0  1  0
 1  0  0

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
6-element BitArray{1}:
 0
```

```
1
0
1
0
0
```

source

Base.trues – Function.

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

Examples

```
julia> trues(2,3)
2×3 BitArray{2}:
 1  1  1
 1  1  1
```

source

Base.falses – Function.

```
falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

```
julia> falses(2,3)
2×3 BitArray{2}:
 0  0  0
 0  0  0
```

source

Base.fill – Function.

```
fill(x, dims::Tuple)
fill(x, dims...)
```

Create an array filled with the value `x`. For example, `fill(1.0, (5,5))` returns a 5×5 array of floats, with each element initialized to `1.0`.

`dims` may be specified as either a tuple or a sequence of arguments. For example, the common idiom `fill(x)` creates a zero-dimensional array containing the single value `x`.

Examples

```
julia> fill(1.0, (2,3))
2×3 Array{Float64,2}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> fill(42)
0-dimensional Array{Int64,0}:
42
```

If `x` is an object reference, all elements will refer to the same object:

```
julia> A = fill(zeros(2), 2);

julia> A[1][1] = 42; # modifies both A[1][1] and A[2][1]

julia> A
2-element Array{Array{Float64,1},1}:
 [42.0, 0.0]
 [42.0, 0.0]
```

source

`Base.fill!` – Function.

```
fill!(A, x)
```

Fill array `A` with the value `x`. If `x` is an object reference, all elements will refer to the same object. `fill!(A, Foo())` will return `A` filled with the result of evaluating `Foo()` once.

Examples

```
julia> A = zeros(2,3)
2×3 Array{Float64,2}:
 0.0  0.0  0.0
```

```
 0.0  0.0  0.0

julia> fill!(A, 2.)
2×3 Array{Float64,2}:
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(undef, 3), a); a[1] = 2; A
3-element Array{Array{Int64,1},1}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(undef, 3), f())
3-element Array{Int64,1}:
 1
 1
 1
```

source

Base.similar – Function.

```
similar(array, [element_type=eltype(array)], [dims=size(array)])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's `eltype` and `size`. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an `Array{element_type}(undef, dims...)`.

For example, `similar(1:10, 1, 4)` returns an uninitialized `Array{Int,2}` since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Array{Int64,2}:
 4419743872  4374413872  4419743888  0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArray`s are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitArray{1}:
 0
 0
```

Since `BitArray`s can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Array{Float64,2}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

source

```
similar(storagetype, axes)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with `axes` specified by the last argument. `storagetype` might be a type or a function.

Examples:

```
similar(Array{Int}, axes(A))
```

creates an array that "acts like" an `Array{Int}` (and might indeed be backed by one), but which is indexed identically to `A`. If `A` has conventional indexing, this will be identical to `Array{Int}(undef, size(A))`, but if `A` has unconventional indexing then the indices of the result will match `A`.

```
similar(BitArray, (axes(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of `A`.

source

51.2 Basic functions

`Base.ndims` – Function.

```
ndims(A::AbstractArray) -> Integer
```

Return the number of dimensions of `A`.

Examples

```
julia> A = fill(1, (3,4,5));

julia> ndims(A)
3
```

source

Base.size – Function.

```
size(A::AbstractArray, [dim])
```

Return a tuple containing the dimensions of A. Optionally you can specify a dimension to just get the length of that dimension.

Note that size may not be defined for arrays with non-standard indices, in which case axes may be useful. See the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (2,3,4));

julia> size(A)
(2, 3, 4)

julia> size(A, 2)
3
```

source

Base.axes – Method.

```
axes(A)
```

Return the tuple of valid indices for array A.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A)
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

source

Base.axes – Method.

```
axes(A, d)
```

Return the valid range of indices for array `A` along dimension `d`.

See also `size`, and the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A, 2)
Base.OneTo(6)
```

source

Base.length – Method.

```
length(A::AbstractArray)
```

Return the number of elements in the array, defaults to `prod(size(A))`.

Examples

```
julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

Base.eachindex – Function.

```
eachindex(A...)
```

Create an iterable object for visiting each index of an `AbstractArray` `A` in an efficient manner. For array types that have opted into fast linear indexing (like `Array`), this is simply the range `1:length(A)`. For other array types, return a specialized Cartesian range to efficiently index into the array with indices specified for every dimension.

For other iterables, including strings and dictionaries, return an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

If you supply more than one `AbstractArray` argument, `eachindex` will create an iterable object that is fast for all arguments (a `UnitRange` if all inputs have fast linear indexing, a `CartesianIndices` otherwise). If the arrays have different sizes and/or dimensionalities, a DimensionMismatch exception will be thrown.

Examples

```
julia> A = [1 2; 3 4];

julia> for i in eachindex(A) # linear indexing
           println(i)
       end
1
2
3
4

julia> for i in eachindex(view(A, 1:2, 1:1)) # Cartesian indexing
           println(i)
       end
CartesianIndex(1, 1)
CartesianIndex(2, 1)
```

source

`Base.IndexStyle` — Type.

```
IndexStyle(A)
IndexStyle(typeof(A))
```

`IndexStyle` specifies the "native indexing style" for array `A`. When you define a new `AbstractArray` type, you can choose to implement either linear indexing (with `IndexLinear`) or cartesian indexing. If you decide to only implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is `IndexCartesian()`.

Julia's internal indexing machinery will automatically (and invisibly) recompute all indexing operations into the preferred style. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your `AbstractArray`, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, `eachindex` creates an iterator whose type depends on the setting of this trait.

source

`Base.IndexLinear` – Type.

```
IndexLinear()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by one linear index.

A linear indexing style uses one integer index to describe the position in the array (even if it's a multidimensional array) and column-major ordering is used to efficiently access the elements. This means that requesting `eachindex` from an array that is `IndexLinear` will return a simple one-dimensional range, even if it is multidimensional.

A custom array that reports its `IndexStyle` as `IndexLinear` only needs to implement indexing (and indexed assignment) with a single `Int` index; all other indexing expressions — including multidimensional accesses — will be recomputed to the linear index. For example, if `A` were a `2×3` custom matrix with linear indexing, and we referenced `A[1, 3]`, this would be recomputed to the equivalent linear index and call `A[5]` since `2*1 + 3 = 5`.

See also `IndexCartesian`.

source

`Base.IndexCartesian` – Type.

```
IndexCartesian()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by a Cartesian index. This is the default for new custom `AbstractArray` subtypes.

A Cartesian indexing style uses multiple integer indices to describe the position in a multidimensional array, with exactly one index per dimension. This means that requesting `eachindex` from an array that is `IndexCartesian` will return a range of `CartesianIndices`.

A `N`-dimensional custom array that reports its `IndexStyle` as `IndexCartesian` needs to implement indexing (and indexed assignment) with exactly `N Int` indices; all other indexing expressions — including linear indexing — will be recomputed to the equivalent Cartesian location. For example, if `A` were a `2×3` custom matrix with cartesian indexing, and we referenced `A[5]`, this would be recomputed to the equivalent Cartesian index and call `A[1, 3]` since `5 = 2*1 + 3`.

It is significantly more expensive to compute Cartesian indices from a linear index than it is to go the other way. The former operation requires division — a very costly operation — whereas the latter only uses multiplication and addition and is essentially free. This asymmetry means it is far more costly to use linear indexing with an `IndexCartesian` array than it is to use Cartesian indexing with an `IndexLinear` array.

See also `IndexLinear`.

source

`Base.conj!` – Function.

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also `conj`.

Examples

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Array{Complex{Int64},2}:
 1+1im  2-1im
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Array{Complex{Int64},2}:
 1-1im  2+1im
 2-2im  3-1im
```

source

`Base.stride` – Function.

```
stride(A, k::Integer)
```

Return the distance in memory (in number of elements) between adjacent elements in dimension `k`.

Examples

```
julia> A = fill(1, (3,4,5));
```

```
julia> stride(A,2)
3

julia> stride(A,3)
12
```

source

Base.strides – Function.

```
strides(A)
```

Return a tuple of the memory strides in each dimension.

Examples

```
julia> A = fill(1, (3,4,5));

julia> strides(A)
(1, 3, 12)
```

source

51.3 Broadcast and vectorization

See also the dot syntax for vectorizing functions; for example, `f.(args...)` implicitly calls `broadcast(f, args...)`. Rather than relying on "vectorized" methods of functions like `sin` to operate on arrays, you should use `sin.(a)` to vectorize via `broadcast`.

Base.Broadcast.broadcast – Function.

```
broadcast(f, As...)
```

Broadcast the function `f` over the arrays, tuples, collections, `Ref`s and/or scalars `As`.

Broadcasting applies the function `f` over the elements of the container arguments and the scalars themselves in `As`. Singleton and missing dimensions are expanded to match the extents of the other arguments by virtually repeating the value. By default, only a limited number of types are considered scalars, including `Number`s, `String`s, `Symbol`s, `Type`s, `Function`s and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

The resulting container type is established by the following rules:

- If all the arguments are scalars or zero-dimensional arrays, it returns an unwrapped scalar.
- If at least one argument is a tuple and all others are scalars or zero-dimensional arrays, it returns a tuple.
- All other combinations of arguments default to returning an `Array`, but custom container types can define their own implementation and promotion-like rules to customize the result when they appear as arguments.

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

Examples

```
julia> A = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
 3   4
 5   6
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Array{Int64,2}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Array{Int64,1}:
 1
 2

julia> abs.((1, -2))
```

```
(1, 2)

julia> broadcast(+, 1.0, (0, -2.0))
(1.0, -1.0)

julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
2-element Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Array{String,1}:
 "one: 1"
 "two: 2"
 "three: 3"
 "four: 4"
```

source

Base.Broadcast.broadcast! – Function.

```
broadcast!(f, dest, As...)
```

Like broadcast, but store the result of `broadcast(f, As...)` in the `dest` array. Note that `dest` is only used to store the result, and does not supply arguments to `f` unless it is also listed in the `As`, as in `broadcast!(f, A, A, B)` to perform `A[:] = broadcast(f, A, B)`.

Examples

```
julia> A = [1.0; 0.0]; B = [0.0; 0.0];

julia> broadcast!(+, B, A, (0, -2.0));

julia> B
2-element Array{Float64,1}:
  1.0
 -2.0

julia> A
2-element Array{Float64,1}:
 1.0
```

```
 0.0

julia> broadcast!(+, A, A, (0, -2.0));

julia> A
2-element Array{Float64,1}:
  1.0
 -2.0
```

source

Base.Broadcast.@__dot__ – Macro.

```
@. expr
```

Convert every function call or operator in `expr` into a "dot call" (e.g. convert `f(x)` to `f.(x)`), and convert every assignment in `expr` to a "dot assignment" (e.g. convert `+=` to `.+=`).

If you want to avoid adding dots for selected function calls in `expr`, splice those function calls in with `$`. For example, `@. sqrt(abs($sort(x)))` is equivalent to `sqrt.(abs.(sort(x)))` (no dot for `sort`).

(`@.` is equivalent to a call to `@__dot__`.)

Examples

```
julia> x = 1.0:3.0; y = similar(x);

julia> @. y = x + 3 * sin(x)
3-element Array{Float64,1}:
 3.5244129544236893
 4.727892280477045
 3.4233600241796016
```

source

For specializing broadcast on custom types, see

Base.Broadcast.BroadcastStyle – Type.

`BroadcastStyle` is an abstract type and trait-function used to determine behavior of objects under broadcasting. `BroadcastStyle(typeof(x))` returns the style associated with `x`. To customize the broadcasting behavior of a type, one can declare a style by defining a type/method pair

```
struct MyContainerStyle <: BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyContainer}) = MyContainerStyle()
```

One then writes method(s) (at least `similar`) operating on `Broadcasted{MyContainerStyle}`. There are also several pre-defined subtypes of `BroadcastStyle` that you may be able to leverage; see the Interfaces chapter for more information.

source

`Base.Broadcast.AbstractArrayStyle` – Type.

`Broadcast.AbstractArrayStyle{N} <: BroadcastStyle` is the abstract supertype for any style associated with an `AbstractArray` type. The `N` parameter is the dimensionality, which can be handy for AbstractArray types that only support specific dimensionalities:

```
struct SparseMatrixStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatrixStyle()
```

For `AbstractArray` types that support arbitrary dimensionality, `N` can be set to `Any`:

```
struct MyArrayStyle <: Broadcast.AbstractArrayStyle{Any} end
Base.BroadcastStyle(::Type{<:MyArray}) = MyArrayStyle()
```

In cases where you want to be able to mix multiple `AbstractArrayStyle`s and keep track of dimensionality, your style needs to support a `Val` constructor:

```
struct MyArrayStyleDim{N} <: Broadcast.AbstractArrayStyle{N} end
(::Type{<:MyArrayStyleDim})(::Val{N}) where N = MyArrayStyleDim{N}()
```

Note that if two or more `AbstractArrayStyle` subtypes conflict, broadcasting machinery will fall back to producing `Array`s. If this is undesirable, you may need to define binary `BroadcastStyle` rules to control the output type.

See also `Broadcast.DefaultArrayStyle`.

source

`Base.Broadcast.ArrayStyle` – Type.

`Broadcast.ArrayStyle{MyArrayType}()` is a `BroadcastStyle` indicating that an object behaves as an array for broadcasting. It presents a simple way to construct `Broadcast.AbstractArrayStyle`s for specific `AbstractArray` container types. Broadcast styles created this way lose track of dimensionality; if keeping track is important for your type, you should create your own custom `Broadcast.AbstractArrayStyle`.

source

Base.Broadcast.DefaultArrayStyle – Type.

> `Broadcast.DefaultArrayStyle{N}()` is a BroadcastStyle indicating that an object behaves as an `N`-dimensional array for broadcasting. Specifically, `DefaultArrayStyle` is used for any `AbstractArray` type that hasn't defined a specialized style, and in the absence of overrides from other `broadcast` arguments the resulting output type is `Array`. When there are multiple inputs to `broadcast`, `DefaultArrayStyle` "loses" to any other Broadcast.ArrayStyle.
>
> source

Base.Broadcast.broadcastable – Function.

```
Broadcast.broadcastable(x)
```

> Return either `x` or an object like `x` such that it supports axes, indexing, and its type supports ndims.
>
> If `x` supports iteration, the returned value should have the same `axes` and indexing behaviors as collect(x).
>
> If `x` is not an `AbstractArray` but it supports `axes`, indexing, and its type supports `ndims`, then `broadcastable(::typeof(x))` may be implemented to just return itself. Further, if `x` defines its own BroadcastStyle, then it must define its `broadcastable` method to return itself for the custom style to have any effect.
>
> Examples

```
julia> Broadcast.broadcastable([1,2,3]) # like `identity` since arrays already support axes and indexing
3-element Array{Int64,1}:
 1
 2
 3

julia> Broadcast.broadcastable(Int) # Types don't support axes, indexing, or iteration but are commonly used as
↪  scalars
Base.RefValue{Type{Int64}}(Int64)

julia> Broadcast.broadcastable("hello") # Strings break convention of matching iteration and act like a scalar
↪  instead
Base.RefValue{String}("hello")
```

> source

Base.Broadcast.combine_axes – Function.

```
combine_axes(As...) -> Tuple
```

Determine the result axes for broadcasting across all values in `As`.

```
julia> Broadcast.combine_axes([1], [1 2; 3 4; 5 6])
(Base.OneTo(3), Base.OneTo(2))

julia> Broadcast.combine_axes(1, 1, 1)
()
```

source

`Base.Broadcast.combine_styles` – Function.

```
combine_styles(cs...) -> BroadcastStyle
```

Decides which `BroadcastStyle` to use for any number of value arguments. Uses `BroadcastStyle` to get the style for each argument, and uses `result_style` to combine styles.

Examples

```
julia> Broadcast.combine_styles([1], [1 2; 3 4])
Base.Broadcast.DefaultArrayStyle{2}()
```

source

`Base.Broadcast.result_style` – Function.

```
result_style(s1::BroadcastStyle[, s2::BroadcastStyle]) -> BroadcastStyle
```

Takes one or two `BroadcastStyle`s and combines them using `BroadcastStyle` to determine a common `BroadcastStyle`.

Examples

```
julia> Broadcast.result_style(Broadcast.DefaultArrayStyle{0}(), Broadcast.DefaultArrayStyle{3}())
Base.Broadcast.DefaultArrayStyle{3}()

julia> Broadcast.result_style(Broadcast.Unknown(), Broadcast.DefaultArrayStyle{1}())
Base.Broadcast.DefaultArrayStyle{1}()
```

source

51.4 Indexing and assignment

Base.getindex – Method.

```
getindex(A, inds...)
```

Return a subset of array A as specified by inds, where each ind may be an Int, an AbstractRange, or a Vector. See the manual section on array indexing for details.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Array{Int64,1}:
 3
 1

julia> getindex(A, 2:4)
3-element Array{Int64,1}:
 3
 2
 4
```

source

Base.setindex! – Method.

```
setindex!(A, X, inds...)
A[inds...] = X
```

Store values from array X within some subset of A as specified by inds. The syntax A[inds...] = X is equivalent to setindex!(A, X, inds...).

Examples

```
julia> A = zeros(2,2);

julia> setindex!(A, [10, 20], [1, 2]);

julia> A[[3, 4]] = [30, 40];

julia> A
2×2 Array{Float64,2}:
 10.0  30.0
 20.0  40.0
```

source

Base.copyto! – Method.

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

source

Base.isassigned – Function.

```
isassigned(array, i) -> Bool
```

Test whether the given array has a value associated with index `i`. Return `false` if the index is out of bounds, or has an undefined reference.

Examples

```
julia> isassigned(rand(3, 3), 5)
true

julia> isassigned(rand(3, 3), 3 * 3 + 1)
false

julia> mutable struct Foo end

julia> v = similar(rand(3), Foo)
3-element Array{Foo,1}:
 #undef
```

```
 #undef
 #undef

julia> isassigned(v, 1)
false
```

source

Base.Colon – Type.

```
Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by to_indices to an internal vector type (Base.Slice) to represent the collection of indices they span before being used.

The singleton instance of Colon is also a function used to construct ranges; see :.

source

Base.IteratorsMD.CartesianIndex – Type.

```
CartesianIndex(i, j, k...)   -> I
CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index I, which can be used for indexing a multidimensional array A. In particular, A[I] is equivalent to A[i,j,k...]. One can freely mix integer and CartesianIndex indices; for example, A[Ipre, i, Ipost] (where Ipre and Ipost are CartesianIndex indices and i is an Int) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A CartesianIndex is sometimes produced by eachindex, and always when iterating with an explicit CartesianIndices.

Examples

```
julia> A = reshape(Vector(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
```

```
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[CartesianIndex((1, 1, 1, 1))]
1

julia> A[CartesianIndex((1, 1, 1, 2))]
9

julia> A[CartesianIndex((1, 1, 2, 1))]
5
```

source

Base.IteratorsMD.CartesianIndices – Type.

```
CartesianIndices(sz::Dims) -> R
CartesianIndices((istart:istop, jstart:jstop, ...)) -> R
```

Define a region `R` spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstop
    for i = istart:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

```
CartesianIndices(A::AbstractArray) -> R
```

As a convenience, constructing a `CartesianIndices` from an array makes a range of its indices.

Examples

```
julia> foreach(println, CartesianIndices((2, 2, 2)))
CartesianIndex(1, 1, 1)
CartesianIndex(2, 1, 1)
CartesianIndex(1, 2, 1)
CartesianIndex(2, 2, 1)
CartesianIndex(1, 1, 2)
CartesianIndex(2, 1, 2)
CartesianIndex(1, 2, 2)
CartesianIndex(2, 2, 2)

julia> CartesianIndices(fill(1, (2,3)))
2×3 CartesianIndices{2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)  CartesianIndex(1, 3)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(2, 3)
```

Conversion between linear and cartesian indices

Linear index to cartesian index conversion exploits the fact that a `CartesianIndices` is an `AbstractArray` and can be indexed linearly:

```
julia> cartesian = CartesianIndices((1:3, 1:2))
3×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)
 CartesianIndex(3, 1)  CartesianIndex(3, 2)

julia> cartesian[4]
CartesianIndex(1, 2)
```

Broadcasting

`CartesianIndices` support broadcasting arithmetic (+ and -) with a `CartesianIndex`.

> Julia 1.1
>
> Broadcasting of CartesianIndices requires at least Julia 1.1.

```
julia> CIs = CartesianIndices((2:3, 5:6))
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 5)  CartesianIndex(2, 6)
 CartesianIndex(3, 5)  CartesianIndex(3, 6)
```

```
julia> CI = CartesianIndex(3, 4)
CartesianIndex(3, 4)

julia> CIs .+ CI
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(5, 9)  CartesianIndex(5, 10)
 CartesianIndex(6, 9)  CartesianIndex(6, 10)
```

For cartesian to linear index conversion, see `LinearIndices`.

source

`Base.Dims` – Type.

```
Dims{N}
```

An `NTuple` of `N` `Int`s used to represent the dimensions of an `AbstractArray`.

source

`Base.LinearIndices` – Type.

```
LinearIndices(A::AbstractArray)
```

Return a `LinearIndices` array with the same shape and `axes` as `A`, holding the linear index of each entry in `A`. Indexing this array with cartesian indices allows mapping them to linear indices.

For arrays with conventional indexing (indices start at 1), or any multidimensional array, linear indices range from 1 to `length(A)`. However, for `AbstractVector`s linear indices are `axes(A, 1)`, and therefore do not start at 1 for vectors with unconventional indexing.

Calling this function is the "safe" way to write algorithms that exploit linear indexing.

Examples

```
julia> A = fill(1, (5,6,7));

julia> b = LinearIndices(A);

julia> extrema(b)
(1, 210)
```

```
LinearIndices(inds::CartesianIndices) -> R
LinearIndices(sz::Dims) -> R
LinearIndices((istart:istop, jstart:jstop, ...)) -> R
```

Return a `LinearIndices` array with the specified shape or `axes`.

Example

The main purpose of this constructor is intuitive conversion from cartesian to linear indexing:

```
julia> linear = LinearIndices((1:3, 1:2))
3×2 LinearIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 1  4
 2  5
 3  6

julia> linear[1,2]
4
```

source

`Base.to_indices` – Function.

```
to_indices(A, I::Tuple)
```

Convert the tuple `I` to a tuple of indices for use in indexing into array `A`.

The returned tuple must only contain either `Int`s or `AbstractArray`s of scalar indices that are supported by array `A`. It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported `Base.to_index(A, i)` to process each index `i`. While this internal function is not intended to be called directly, `Base.to_index` may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, `to_indices(A, I)` calls `to_indices(A, axes(A), I)`, which then recursively walks through both the given tuple of indices and the dimensional indices of `A` in tandem. As such, not all index types are guaranteed to propagate to `Base.to_index`.

source

`Base.checkbounds` – Function.

```
checkbounds(Bool, A, I...)
```

Return `true` if the specified indices `I` are in bounds for the given array `A`. Subtypes of `AbstractArray` should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on `A`'s indices and `checkindex`.

See also `checkindex`.

Examples

```
julia> A = rand(3, 3);

julia> checkbounds(Bool, A, 2)
true

julia> checkbounds(Bool, A, 3, 4)
false

julia> checkbounds(Bool, A, 1:3)
true

julia> checkbounds(Bool, A, 1:3, 2:4)
false
```

source

```
checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

source

`Base.checkindex` – Function.

```
checkindex(Bool, inds::AbstractUnitRange, index)
```

Return `true` if the given `index` is within the bounds of `inds`. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

Examples

```
julia> checkindex(Bool, 1:20, 8)
true
```

```
julia> checkindex(Bool, 1:20, 21)
false
```

source

51.5 Views (SubArrays and other view types)

Base.view – Function.

```
view(A, inds...)
```

Like getindex, but returns a view into the parent array A with the given indices instead of making a copy. Calling getindex or setindex! on the returned SubArray computes the indices to the parent array on the fly without checking bounds.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Array{Int64,2}:
 0  2
 0  4
```

source

Base.@view – Macro.

```
@view A[inds...]
```

Creates a `SubArray` from an indexing expression. This can only be applied directly to a reference expression (e.g. `@view A[1,2:end]`), and should not be used as the target of an assignment (e.g. `@view(A[1,2:end]) = ...`). See also `@views` to switch an entire block of code to use views for slicing.

> Julia 1.5
>
> Using `begin` in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Array{Int64,2}:
 0  2
 0  4
```

source

`Base.@views` — Macro.

```
@views expression
```

Convert every array-slicing operation in the given expression (which may be a `begin`/`end` block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit `getindex` calls (as opposed to `array[...]`) are unaffected.

Note

The `@views` macro only affects `array[...]` expressions that appear explicitly in the given `expression`, not array slicing that occurs in functions called by that code.

Julia 1.5

Using `begin` in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = zeros(3, 3);

julia> @views for row in 1:3
           b = A[row, :]
           b[:] .= row
       end

julia> A
3×3 Array{Float64,2}:
 1.0  1.0  1.0
 2.0  2.0  2.0
 3.0  3.0  3.0
```

source

`Base.parent` – Function.

```
parent(A)
```

Return the underlying "parent array". This parent array of objects of types `SubArray`, `ReshapedArray` or `LinearAlgebra.Transpose` is what was passed as an argument to `view`, `reshape`, `transpose`, etc. during object creation. If the input is not a wrapped object, return the input itself.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> V = view(A, 1:2, :)
```

```
2×2 view(::Array{Int64,2}, 1:2, :) with eltype Int64:
 1  2
 3  4

julia> parent(V)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.parentindices – Function.

```
parentindices(A)
```

Return the indices in the parent which correspond to the array view A.

Examples

```
julia> A = [1 2; 3 4];

julia> V = view(A, 1, :)
2-element view(::Array{Int64,2}, 1, :) with eltype Int64:
 1
 2

julia> parentindices(V)
(1, Base.Slice(Base.OneTo(2)))
```

source

Base.selectdim – Function.

```
selectdim(A, d::Integer, i)
```

Return a view of all the data of A where the index for dimension d equals i.

Equivalent to view(A,:,:,...,i,:,:,...) where i is in position d.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8

julia> selectdim(A, 2, 3)
2-element view(::Array{Int64,2}, :, 3) with eltype Int64:
 3
 7
```

source

Base.reinterpret – Function.

```
reinterpret(type, A)
```

Change the type-interpretation of a block of memory. For arrays, this constructs a view of the array with the same binary data as the given array, but with the specified element type. For example, `reinterpret(Float32, UInt32(7))` interprets the 4 bytes corresponding to `UInt32(7)` as a `Float32`.

Examples

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 reinterpret(Float32, ::Array{UInt32,2}):
 1.0f-45  3.0f-45  4.0f-45  6.0f-45  7.0f-45
```

source

Base.reshape – Function.

```
reshape(A, dims...) -> AbstractArray
reshape(A, dims) -> AbstractArray
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a `:`, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array `A`. The total number of elements must not change.

Examples

```
julia> A = Vector(1:16)
16-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Array{Int64,2}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16

julia> reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6
```

source

Base.dropdims – Function.

```
dropdims(A; dims)
```

Remove the dimensions specified by `dims` from array `A`. Elements of `dims` must be unique and within the range `1:ndims(A)`. `size(A,i)` must equal 1 for all `i` in `dims`.

Examples

```
julia> a = reshape(Vector(1:4),(2,2,1,1))
2×2×1×1 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

julia> dropdims(a; dims=3)
2×2×1 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4
```

source

`Base.vec` – Function.

```
vec(a::AbstractArray) -> AbstractVector
```

Reshape the array `a` as a one-dimensional column vector. Return `a` if it is already an `AbstractVector`. The resulting array shares the same underlying data as `a`, so it will only be mutable if `a` is mutable, in which case modifying one will also modify the other.

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Array{Int64,1}:
 1
 4
 2
```

```
 5
 3
 6


julia> vec(1:3)
1:3
```

See also reshape.

source

51.6 Concatenation and permutation

Base.cat – Function.

```
cat(A...; dims=dims)
```

Concatenate the input arrays along the specified dimensions in the iterable **dims**. For dimensions not in **dims**, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in **dims**, the size of the output array is the sum of the sizes of the input arrays along that dimension. If **dims** is a single number, the different arrays are tightly stacked along that dimension. If **dims** is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, **cat(matrices...; dims=(1,2))** builds a block diagonal matrix, i.e. a block matrix with **matrices[1]**, **matrices[2]**, ... as diagonal blocks and matching zero blocks away from the diagonal.

source

Base.vcat – Function.

```
vcat(A...)
```

Concatenate along dimension 1.

Examples

```
julia> a = [1 2 3 4 5]
1×5 Array{Int64,2}:
 1  2  3  4  5


julia> b = [6 7 8 9 10; 11 12 13 14 15]
```

```
2×5 Array{Int64,2}:
  6   7   8   9  10
 11  12  13  14  15

julia> vcat(a,b)
3×5 Array{Int64,2}:
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15

julia> c = ([1 2 3], [4 5 6])
([1 2 3], [4 5 6])

julia> vcat(c...)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

source

Base.hcat – Function.

```
hcat(A...)
```

Concatenate along dimension 2.

Examples

```
julia> a = [1; 2; 3; 4; 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> b = [6 7; 8 9; 10 11; 12 13; 14 15]
5×2 Array{Int64,2}:
  6   7
  8   9
 10  11
```

```
 12  13
 14  15

julia> hcat(a,b)
5×3 Array{Int64,2}:
 1   6   7
 2   8   9
 3  10  11
 4  12  13
 5  14  15

julia> c = ([1; 2; 3], [4; 5; 6])
([1, 2, 3], [4, 5, 6])

julia> hcat(c...)
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6

julia> x = Matrix(undef, 3, 0)  # x = [] would have created an Array{Any, 1}, but need an Array{Any, 2}
3×0 Array{Any,2}

julia> hcat(x, [1; 2; 3])
3×1 Array{Any,2}:
 1
 2
 3
```

source

Base.hvcat — Function.

```
hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)
```

```
julia> [a b c; d e f]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> [a b;c d; e f]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

If the first argument is a single integer `n`, then all block rows are assumed to have `n` block columns.

source

`Base.vect` – Function.

```
vect(X...)
```

Create a `Vector` with element type computed from the `promote_typeof` of the argument, containing the argument list.

Examples

```
julia> a = Base.vect(UInt8(1), 2.5, 1//2)
3-element Array{Float64,1}:
 1.0
 2.5
 0.5
```

source

Base.circshift – Function.

```
circshift(A, shifts)
```

Circularly shift, i.e. rotate, the data in an array. The second argument is a tuple or vector giving the amount to shift in each dimension, or an integer to shift only in the first dimension.

Examples

```
julia> b = reshape(Vector(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> circshift(b, (0,2))
4×4 Array{Int64,2}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
4×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15
 4  8  12  16
 1  5   9  13

julia> a = BitArray([true, true, false, false, true])
5-element BitArray{1}:
 1
 1
 0
 0
 1

julia> circshift(a, 1)
```

```
5-element BitArray{1}:
 1
 1
 1
 0
 0

julia> circshift(a, -1)
5-element BitArray{1}:
 1
 0
 0
 1
 1
```

See also circshift!.

source

Base.circshift! – Function.

```
circshift!(dest, src, shifts)
```

Circularly shift, i.e. rotate, the data in src, storing the result in dest. shifts specifies the amount to shift in each dimension.

The dest array must be distinct from the src array (they cannot alias each other).

See also circshift.

source

Base.circcopy! – Function.

```
circcopy!(dest, src)
```

Copy src to dest, indexing each dimension modulo its length. src and dest must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap dest agrees with src.

Examples

```
julia> src = reshape(Vector(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> dest = OffsetArray{Int}(undef, (0:3,2:5))

julia> circcopy!(dest, src)
OffsetArrays.OffsetArray{Int64,2,Array{Int64,2}} with indices 0:3×2:5:
 8  12  16  4
 5   9  13  1
 6  10  14  2
 7  11  15  3

julia> dest[1:3,2:4] == src[1:3,2:4]
true
```

source

Base.findall – Method.

```
findall(A)
```

Return a vector `I` of the `true` indices or keys of `A`. If there are no such elements of `A`, return an empty array. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [true, false, false, true]
4-element Array{Bool,1}:
 1
 0
 0
 1

julia> findall(A)
2-element Array{Int64,1}:
```

```
 1
 4

julia> A = [true false; false true]
2×2 Array{Bool,2}:
 1  0
 0  1

julia> findall(A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 2)

julia> findall(falses(3))
Int64[]
```

source

Base.findall – Method.

```
findall(f::Function, A)
```

Return a vector `I` of the indices or keys of `A` where `f(A[I])` returns `true`. If there are no such elements of `A`, return an empty array.

Indices or keys are of the same type as those returned by keys(A) and pairs(A).

Examples

```
julia> x = [1, 3, 4]
3-element Array{Int64,1}:
 1
 3
 4

julia> findall(isodd, x)
2-element Array{Int64,1}:
 1
 2

julia> A = [1 2 0; 3 4 0]
```

```
2×3 Array{Int64,2}:
 1  2  0
 3  4  0
julia> findall(isodd, A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)

julia> findall(!iszero, A)
4-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)

julia> d = Dict(:A => 10, :B => -1, :C => 0)
Dict{Symbol,Int64} with 3 entries:
  :A => 10
  :B => -1
  :C => 0

julia> findall(x -> x >= 0, d)
2-element Array{Symbol,1}:
 :A
 :C
```

source

Base.findfirst – Method.

```
findfirst(A)
```

Return the index or key of the first `true` value in `A`. Return `nothing` if no such value is found. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Array{Bool,1}:
```

```
 0
 0
 1
 0

julia> findfirst(A)
3

julia> findfirst(falses(3)) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Array{Bool,2}:
 0  0
 1  0

julia> findfirst(A)
CartesianIndex(2, 1)
```

source

Base.findfirst – Method.

```
findfirst(predicate::Function, A)
```

Return the index or key of the first element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by keys(A) and pairs(A).

Examples

```
julia> A = [1, 4, 2, 2]
4-element Array{Int64,1}:
 1
 4
 2
 2

julia> findfirst(iseven, A)
2
```

```
julia> findfirst(x -> x>10, A) # returns nothing, but not printed in the REPL

julia> findfirst(isequal(4), A)
2

julia> A = [1 4; 2 2]
2×2 Array{Int64,2}:
 1  4
 2  2

julia> findfirst(iseven, A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(A)
```

Return the index or key of the last `true` value in `A`. Return `nothing` if there is no `true` value in `A`.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [true, false, true, false]
4-element Array{Bool,1}:
 1
 0
 1
 0

julia> findlast(A)
3

julia> A = falses(2,2);

julia> findlast(A) # returns nothing, but not printed in the REPL

julia> A = [true false; true false]
2×2 Array{Bool,2}:
```

```
 1  0
 1  0

julia> findlast(A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(predicate::Function, A)
```

Return the index or key of the last element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> findlast(isodd, A)
3

julia> findlast(x -> x > 5, A) # returns nothing, but not printed in the REPL

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> findlast(isodd, A)
CartesianIndex(2, 1)
```

source

Base.findnext – Method.

```
findnext(A, i)
```

Find the next index after or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Array{Bool,1}:
 0
 0
 1
 0

julia> findnext(A, 1)
3

julia> findnext(A, 4) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Array{Bool,2}:
 0  0
 1  0

julia> findnext(A, CartesianIndex(1, 1))
CartesianIndex(2, 1)
```

source

`Base.findnext` – Method.

```
findnext(predicate::Function, A, i)
```

Find the next index after or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2];
```

```
julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2) # returns nothing, but not printed in the REPL

julia> A = [1 4; 2 2];

julia> findnext(isodd, A, CartesianIndex(1, 1))
CartesianIndex(1, 1)
```

source

Base.findprev – Method.

```
findprev(A, i)
```

Find the previous index before or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by keys(A) and pairs(A).

Examples

```
julia> A = [false, false, true, true]
4-element Array{Bool,1}:
 0
 0
 1
 1

julia> findprev(A, 3)
3

julia> findprev(A, 1) # returns nothing, but not printed in the REPL

julia> A = [false false; true true]
2×2 Array{Bool,2}:
 0  0
 1  1

julia> findprev(A, CartesianIndex(2, 1))
CartesianIndex(2, 1)
```

source

Base.findprev – Method.

```
findprev(predicate::Function, A, i)
```

Find the previous index before or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [4, 6, 1, 2]
4-element Array{Int64,1}:
 4
 6
 1
 2

julia> findprev(isodd, A, 1) # returns nothing, but not printed in the REPL

julia> findprev(isodd, A, 3)
3

julia> A = [4 6; 1 2]
2×2 Array{Int64,2}:
 4  6
 1  2

julia> findprev(isodd, A, CartesianIndex(1, 2))
CartesianIndex(2, 1)
```

source

Base.permutedims – Function.

```
permutedims(A::AbstractArray, perm)
```

Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`.

See also: `PermutedDimsArray`.

Examples

```
julia> A = reshape(Vector(1:8), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 5  7
 6  8

julia> permutedims(A, [3, 2, 1])
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 5  7

[:, :, 2] =
 2  4
 6  8
```

source

```
permutedims(m::AbstractMatrix)
```

Permute the dimensions of the matrix `m`, by flipping the elements across the diagonal of the matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> a = [1 2; 3 4];

julia> b = [5 6; 7 8];

julia> c = [9 10; 11 12];

julia> d = [13 14; 15 16];

julia> X = [[a] [b]; [c] [d]]
2×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]     [5 6; 7 8]
 [9 10; 11 12]  [13 14; 15 16]
```

```
julia> permutedims(X)
2×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [9 10; 11 12]
 [5 6; 7 8]  [13 14; 15 16]

julia> transpose(X)
2×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},2}}:
 [1 3; 2 4]  [9 11; 10 12]
 [5 7; 6 8]  [13 15; 14 16]
```

source

```
permutedims(v::AbstractVector)
```

Reshape vector `v` into a `1 × length(v)` row matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> permutedims([1, 2, 3, 4])
1×4 Array{Int64,2}:
 1  2  3  4

julia> V = [[[1 2; 3 4]]; [[5 6; 7 8]]]
2-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
 [5 6; 7 8]

julia> permutedims(V)
1×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [5 6; 7 8]

julia> transpose(V)
1×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},1}}:
 [1 3; 2 4]  [5 7; 6 8]
```

source

`Base.permutedims!` – Function.

```
permutedims!(dest, src, perm)
```

Permute the dimensions of array `src` and store the result in the array `dest`. `perm` is a vector specifying a permutation of length `ndims(src)`. The preallocated array `dest` should have `size(dest) == size(src)[perm]` and is completely overwritten. No in-place permutation is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

See also permutedims.

source

Base.PermutedDimsArrays.PermutedDimsArray – Type.

```
PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray `A`, create a view `B` such that the dimensions appear to be permuted. Similar to `permutedims`, except that no copying occurs (`B` shares storage with `A`).

See also: permutedims.

Examples

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));

julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

source

Base.promote_shape – Function.

```
promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

Examples

```
julia> a = fill(1, (3,4,1,1,1));
```

```
julia> b = fill(1, (3,4));

julia> promote_shape(a,b)
(Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1), Base.OneTo(1))

julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))
(2, 3, 1, 4, 1)
```

source

51.7 Array functions

Base.accumulate – Function.

```
accumulate(op, A; dims::Integer, [init])
```

Cumulative operation `op` along the dimension `dims` of `A` (providing `dims` is optional for vectors). An initial value `init` may optionally be provided by a keyword argument. See also accumulate! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow). For common operations there are specialized variants of `accumulate`, see: cumsum, cumprod

> Julia 1.5
>
> `accumulate` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> accumulate(+, [1,2,3])
3-element Array{Int64,1}:
 1
 3
 6

julia> accumulate(*, [1,2,3])
3-element Array{Int64,1}:
 1
 2
 6

julia> accumulate(+, [1,2,3]; init=100)
3-element Array{Int64,1}:
```

```
 101
 103
 106

julia> accumulate(min, [1,2,-1]; init=0)
3-element Array{Int64,1}:
  0
  0
 -1

julia> accumulate(+, fill(1, 3, 3), dims=1)
3×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3

julia> accumulate(+, fill(1, 3, 3), dims=2)
3×3 Array{Int64,2}:
 1  2  3
 1  2  3
 1  2  3
```

source

Base.accumulate! – Function.

```
accumulate!(op, B, A; [dims], [init])
```

Cumulative operation `op` on `A` along the dimension `dims`, storing the result in `B`. Providing `dims` is optional for vectors. If the keyword argument `init` is given, its value is used to instantiate the accumulation. See also accumulate.

Examples

```
julia> x = [1, 0, 2, 0, 3];

julia> y = [0, 0, 0, 0, 0];

julia> accumulate!(+, y, x);

julia> y
```

```
5-element Array{Int64,1}:
 1
 1
 3
 3
 6

julia> A = [1 2; 3 4];

julia> B = [0 0; 0 0];

julia> accumulate!(-, B, A, dims=1);

julia> B
2×2 Array{Int64,2}:
  1   2
 -2  -2

julia> accumulate!(-, B, A, dims=2);

julia> B
2×2 Array{Int64,2}:
 1  -1
 3  -1
```

source

Base.cumprod – Function.

```
cumprod(A; dims::Integer)
```

Cumulative product along the dimension **dim**. See also cumprod! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

```
julia> cumprod(a, dims=1)
2×3 Array{Int64,2}:
 1   2   3
 4  10  18

julia> cumprod(a, dims=2)
2×3 Array{Int64,2}:
 1   2    6
 4  20  120
```

source

```
cumprod(itr)
```

Cumulative product of an iterator. See also `cumprod!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

> Julia 1.5
>
> `cumprod` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumprod(fill(1//2, 3))
3-element Array{Rational{Int64},1}:
 1//2
 1//4
 1//8

julia> cumprod([fill(1//3, 2, 2) for i in 1:3])
3-element Array{Array{Rational{Int64},2},1}:
 [1//3 1//3; 1//3 1//3]
 [2//9 2//9; 2//9 2//9]
 [4//27 4//27; 4//27 4//27]

julia> cumprod((1, 2, 1))
(1, 2, 2)

julia> cumprod(x^2 for x in 1:3)
3-element Array{Int64,1}:
  1
```

```
 4
36
```

source

Base.cumprod! – Function.

```
cumprod!(B, A; dims::Integer)
```

Cumulative product of A along the dimension dims, storing the result in B. See also cumprod.

source

```
cumprod!(y::AbstractVector, x::AbstractVector)
```

Cumulative product of a vector x, storing the result in y. See also cumprod.

source

Base.cumsum – Function.

```
cumsum(A; dims::Integer)
```

Cumulative sum along the dimension dims. See also cumsum! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Array{Int64,2}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Array{Int64,2}:
 1  3   6
 4  9  15
```

Note

The return array's `eltype` is `Int` for signed integers of less than system word size and `UInt` for unsigned integers of less than system word size. To preserve `eltype` of arrays with small signed or unsigned integer `accumulate(+, A)` should be used.

```
julia> cumsum(Int8[100, 28])
2-element Array{Int64,1}:
 100
 128

julia> accumulate(+,Int8[100, 28])
2-element Array{Int8,1}:
  100
 -128
```

In the former case, the integers are widened to system word size and therefore the result is `Int64[100, 128]`. In the latter case, no such widening happens and integer overflow results in `Int8[100, -128]`.

source

```
cumsum(itr)
```

Cumulative sum an iterator. See also `cumsum!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Julia 1.5

`cumsum` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumsum([1, 1, 1])
3-element Array{Int64,1}:
 1
 2
 3

julia> cumsum([fill(1, 2) for i in 1:3])
3-element Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]
 [3, 3]
```

```
julia> cumsum((1, 1, 1))
(1, 2, 3)

julia> cumsum(x^2 for x in 1:3)
3-element Array{Int64,1}:
  1
  5
 14
```

source

Base.cumsum! – Function.

```
cumsum!(B, A; dims::Integer)
```

Cumulative sum of A along the dimension dims, storing the result in B. See also cumsum.

source

Base.diff – Function.

```
diff(A::AbstractVector)
diff(A::AbstractArray; dims::Integer)
```

Finite difference operator on a vector or a multidimensional array A. In the latter case the dimension to operate on needs to be specified with the dims keyword argument.

> Julia 1.1
>
> diff for arrays with dimension higher than 2 requires at least Julia 1.1.

Examples

```
julia> a = [2 4; 6 16]
2×2 Array{Int64,2}:
 2   4
 6  16

julia> diff(a, dims=2)
2×1 Array{Int64,2}:
  2
```

```
 10


julia> diff(vec(a))
3-element Array{Int64,1}:
  4
 -2
 12
```

source

Base.repeat – Function.

```
repeat(A::AbstractArray, counts::Integer...)
```

Construct an array by repeating array `A` a given number of times in each dimension, specified by `counts`.

Examples

```
julia> repeat([1, 2, 3], 2)
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3


julia> repeat([1, 2, 3], 2, 3)
6×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

source

```
repeat(A::AbstractArray; inner=ntuple(x->1, ndims(A)), outer=ntuple(x->1, ndims(A)))
```

Construct an array by repeating the entries of `A`. The i-th element of `inner` specifies the number of times that the individual entries of the i-th dimension of `A` should be repeated. The i-th element of `outer` specifies the number

of times that a slice along the i-th dimension of `A` should be repeated. If `inner` or `outer` are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Array{Int64,1}:
 1
 1
 2
 2

julia> repeat(1:2, outer=2)
4-element Array{Int64,1}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Array{Int64,2}:
 1  2  1  2  1  2
 1  2  1  2  1  2
 3  4  3  4  3  4
 3  4  3  4  3  4
```

source

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also: `^`

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

`Base.rot180` – Function.

```
rot180(A)
```

Rotate matrix `A` 180 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a)
2×2 Array{Int64,2}:
 4  3
 2  1
```

source

```
rot180(A, k)
```

Rotate matrix `A` 180 degrees an integer `k` number of times. If `k` is even, this is equivalent to a `copy`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a,1)
2×2 Array{Int64,2}:
```

```
 4  3
 2  1

julia> rot180(a,2)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.rotl90 – Function.

```
rotl90(A)
```

Rotate matrix A left 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotl90(a)
2×2 Array{Int64,2}:
 2  4
 1  3
```

source

```
rotl90(A, k)
```

Left-rotate matrix A 90 degrees counterclockwise an integer k number of times. If k is a multiple of four (including zero), this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> rotl90(a,1)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotl90(a,3)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.rotr90 – Function.

```
rotr90(A)
```

Rotate matrix `A` right 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotr90(a)
2×2 Array{Int64,2}:
 3  1
 4  2
```

source

```
rotr90(A, k)
```

Right-rotate matrix `A` 90 degrees clockwise an integer `k` number of times. If `k` is a multiple of four (including zero), this is equivalent to a `copy`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotr90(a,1)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

`Base.mapslices` – Function.

```
mapslices(f, A; dims)
```

Transform the given dimensions of array `A` using function `f`. `f` is called on each slice of `A` of the form `A[...,:,...,:,...]`. `dims` is an integer vector specifying where the colons go in this expression. The results are concatenated along

the remaining dimensions. For example, if `dims` is `[1,2]` and `A` is 4-dimensional, `f` is called on `A[:,:,i,j]` for all `i` and `j`.

Examples

```
julia> a = reshape(Vector(1:16),(2,2,2,2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> mapslices(sum, a, dims = [1,2])
1×1×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 10

[:, :, 2, 1] =
 26

[:, :, 1, 2] =
 42

[:, :, 2, 2] =
 58
```

source

51.8 Combinatorics

Base.invperm – Function.

```
invperm(v)
```

Return the inverse permutation of v. If B = A[v], then A == B[invperm(v)].

Examples

```
julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Array{Int64,1}:
 4
 1
 3
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Array{Char,1}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> B[invperm(v)]
4-element Array{Char,1}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

source

Base.isperm – Function.

```
isperm(v) -> Bool
```

Return true if v is a valid permutation.

Examples

```
julia> isperm([1; 2])
true

julia> isperm([1; 3])
false
```

source

Base.permute! – Method.

```
permute!(v, p)
```

Permute vector `v` in-place, according to permutation `p`. No checking is done to verify that `p` is a permutation.

To return a new permutation, use `v[p]`. Note that this is generally faster than `permute!(v,p)` for large vectors.

See also invpermute!.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Array{Int64,1}:
 1
 4
 3
 1
```

source

Base.invpermute! – Function.

```
invpermute!(v, p)
```

Like permute!, but the inverse of the given permutation is applied.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> invpermute!(A, perm);

julia> A
4-element Array{Int64,1}:
 4
 1
 3
 1
```

source

Base.reverse – Method.

```
reverse(v [, start=1 [, stop=length(v) ]] )
```

Return a copy of v reversed from start to stop. See also Iterators.reverse for reverse-order iteration without making a copy.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse(A)
5-element Array{Int64,1}:
 5
 4
 3
 2
 1

julia> reverse(A, 1, 4)
```

```
5-element Array{Int64,1}:
 4
 3
 2
 1
 5

julia> reverse(A, 3, 5)
5-element Array{Int64,1}:
 1
 2
 5
 4
 3
```

source

```
reverse(A; dims::Integer)
```

Reverse `A` in dimension `dims`.

Examples

```
julia> b = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> reverse(b, dims=2)
2×2 Array{Int64,2}:
 2  1
 4  3
```

source

`Base.reverseind` – Function.

```
reverseind(v, i)
```

Given an index `i` in `reverse(v)`, return the corresponding index in `v` so that `v[reverseind(v,i)] == reverse(v)[i]`. (This can be nontrivial in cases where `v` contains non-ASCII characters.)

Examples

```
julia> r = reverse("Julia")
"ailuJ"

julia> for i in 1:length(r)
           print(r[reverseind("Julia", i)])
       end
Julia
```

source

Base.reverse! – Function.

```
reverse!(v [, start=1 [, stop=length(v) ]]) -> v
```

In-place version of reverse.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse!(A);

julia> A
5-element Array{Int64,1}:
 5
 4
 3
 2
 1
```

source

Chapter 52

Tasks

Core.Task – Type.

```
Task(func)
```

Create a `Task` (i.e. coroutine) to execute the given function `func` (which must be callable with no arguments). The task exits when this function returns.

Examples

```
julia> a() = sum(i for i in 1:1000);

julia> b = Task(a);
```

In this example, `b` is a runnable `Task` that hasn't started yet.

source

Base.@task – Macro.

```
@task
```

Wrap an expression in a Task without executing it, and return the Task. This only creates a task, and does not run it.

Examples

```
julia> a1() = sum(i for i in 1:1000);

julia> b = @task a1();
```

```
julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

Base.@async – Macro.

```
@async
```

Wrap an expression in a Task and add it to the local machine's scheduler queue.

Values can be interpolated into `@async` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the value of a variable, isolating the aysnchronous code from changes to the variable's value in the current task.

> Julia 1.4
>
> Interpolating values via `$` is available as of Julia 1.4.

source

Base.@sync – Macro.

```
@sync
```

Wait until all lexically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@distributed` are complete. All exceptions thrown by enclosed async operations are collected and thrown as a `CompositeException`.

source

Base.asyncmap – Function.

```
asyncmap(f, c...; ntasks=0, batch_size=nothing)
```

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func` is less than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `objectid` of the tasks in which the mapping function is executed.

First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = objectid(current_task());

julia> asyncmap(x->tskoid(), 1:5)
5-element Array{UInt64,1}:
 0x6e15e66c75c75853
 0x440f8819a1baa682
 0x9fb3eeadd0c83985
 0xebd3e35fe90d4050
 0x29efc93edce2b961

julia> length(unique(asyncmap(x->tskoid(), 1:5)))
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)
5-element Array{UInt64,1}:
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94

julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))
```

```
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", element_val: ", x[1], ", task: ", tskoid()),
     input)
batch_func (generic function with 1 method)

julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Array{String,1}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
 "args_tuple: (4,), element_val: 4, task: 4904288162898683522"
 "args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

> Note
>
> Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, `asyncmap` is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

source

`Base.asyncmap!` – Function.

```
asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like `asyncmap`, but stores output in `results` rather than returning a collection.

source

`Base.fetch` – Method.

```
fetch(t::Task)
```

Wait for a Task to finish, then return its result value. If the task fails with an exception, a `TaskFailedException` (which wraps the failed task) is thrown.

source

`Base.current_task` – Function.

```
current_task()
```

Get the currently running Task.

source

Base.istaskdone – Function.

```
istaskdone(t::Task) -> Bool
```

Determine whether a task has exited.

Examples

```
julia> a2() = sum(i for i in 1:1000);

julia> b = Task(a2);

julia> istaskdone(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

Base.istaskstarted – Function.

```
istaskstarted(t::Task) -> Bool
```

Determine whether a task has started executing.

Examples

```
julia> a3() = sum(i for i in 1:1000);

julia> b = Task(a3);

julia> istaskstarted(b)
false
```

source

Base.istaskfailed – Function.

```
istaskfailed(t::Task) -> Bool
```

Determine whether a task has exited because an exception was thrown.

Examples

```
julia> a4() = error("task failed");

julia> b = Task(a4);

julia> istaskfailed(b)
false

julia> schedule(b);

julia> yield();

julia> istaskfailed(b)
true
```

source

Base.task_local_storage – Method.

```
task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(key, value)
```

Assign a value to a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(body, key, value)
```

Call the function `body` with a modified task-local storage, in which `value` is assigned to `key`; the previous value of `key`, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

source

Chapter 53

Scheduling

`Base.yield` — Function.

```
yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

source

```
yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of `schedule(t, arg); yield()` which immediately yields to `t` before calling the scheduler.

source

`Base.yieldto` — Function.

```
yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

source

`Base.sleep` — Function.

```
sleep(seconds)
```

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

source

`Base.wait` – Function.

```
wait(r::Future)
```

Wait for a value to become available for the specified `Future`.

```
wait(r::RemoteChannel, args...)
```

Wait for a value to become available on the specified `RemoteChannel`.

```
wait([x])
```

Block the current task until some event occurs, depending on the type of the argument:

- `Channel`: Wait for a value to be appended to the channel.
- `Condition`: Wait for `notify` on a condition.
- `Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.
- `Task`: Wait for a `Task` to finish. If the task fails with an exception, a `TaskFailedException` (which wraps the failed task) is thrown.
- `RawFD`: Wait for changes on a file descriptor (see the `FileWatching` package).

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

source

Special note for `Threads.Condition`:

The caller must be holding the `lock` that owns `c` before calling this method. The calling task will be blocked until some other task wakes it, usually by calling `notify` on the same Condition object. The lock will be atomically released when blocking (even if it was locked recursively), and will be reacquired before returning.

source

`Base.timedwait` – Function.

```
timedwait(testcb::Function, timeout::Real; pollint::Real=0.1)
```

Waits until `testcb` returns `true` or for `timeout` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds. The minimum duration for `timeout` and `pollint` is 1 millisecond or `0.001`.

Returns :ok or :timed_out

source

`Base.Condition` – Type.

```
Condition()
```

Create an edge-triggered event source that tasks can wait for. Tasks that call `wait` on a `Condition` are suspended and queued. Tasks are woken up when `notify` is later called on the `Condition`. Edge triggering means that only tasks waiting at the time `notify` is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The `Channel` and `Threads.Event` types do this, and can be used for level-triggered events.

This object is NOT thread-safe. See `Threads.Condition` for a thread-safe version.

source

Missing docstring.

Missing docstring for `Base.Threads.Condition`. Check Documenter's build log for details.

`Base.notify` – Function.

```
notify(condition, val=nothing; all=true, error=false)
```

Wake up tasks waiting for a condition, passing them `val`. If `all` is `true` (the default), all waiting tasks are woken, otherwise only one is. If `error` is `true`, the passed value is raised as an exception in the woken tasks.

Return the count of tasks woken up. Return 0 if no tasks are waiting on `condition`.

source

`Base.schedule` – Function.

```
schedule(t::Task, [val]; error=false)
```

Add a Task to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as wait.

If a second argument `val` is provided, it will be passed to the task (via the return value of yieldto) when it runs again. If `error` is `true`, the value is raised as an exception in the woken task.

Examples

```
julia> a5() = sum(i for i in 1:1000);

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
true

julia> istaskdone(b)
true
```

source

Missing docstring.

Missing docstring for `Base.Threads.Event`. Check Documenter's build log for details.

Base.Semaphore – Type.

```
Semaphore(sem_size)
```

Create a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be matched with a release.

source

Base.acquire – Function.

```
acquire(s::Semaphore)
```

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

source

`Base.release` – Function.

```
release(s::Semaphore)
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

source

`Base.AbstractLock` – Type.

```
AbstractLock
```

Abstract supertype describing types that implement the synchronization primitives: `lock`, `trylock`, `unlock`, and `islocked`.

source

`Base.lock` – Function.

```
lock(lock)
```

Acquire the `lock` when it becomes available. If the lock is already locked by a different task/thread, wait for it to become available.

Each `lock` must be matched by an `unlock`.

source

```
lock(f::Function, lock)
```

Acquire the `lock`, execute `f` with the `lock` held, and release the `lock` when `f` returns. If the lock is already locked by a different task/thread, wait for it to become available.

When this function returns, the `lock` has been released, so the caller should not attempt to `unlock` it.

source

`Base.unlock` – Function.

```
unlock(lock)
```

Releases ownership of the `lock`.

If this is a recursive lock which has been acquired before, decrement an internal counter and return immediately.

source

`Base.trylock` – Function.

```
trylock(lock) -> Success (Boolean)
```

Acquire the lock if it is available, and return `true` if successful. If the lock is already locked by a different task/thread, return `false`.

Each successful `trylock` must be matched by an `unlock`.

source

`Base.islocked` – Function.

```
islocked(lock) -> Status (Boolean)
```

Check whether the `lock` is held by any task/thread. This should not be used for synchronization (see instead `trylock`).

source

`Base.ReentrantLock` – Type.

```
ReentrantLock()
```

Creates a re-entrant lock for synchronizing `Task`s. The same task can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

source

`Base.Channel` – Type.

```
Channel{T=Any}(size::Int=0)
```

Constructs a `Channel` with an internal buffer that can hold a maximum of `size` objects of type `T`. `put!` calls on a full channel block until an object is removed with `take!`.

`Channel(0)` constructs an unbuffered channel. `put!` blocks until a matching `take!` is called. And vice-versa.

Other constructors:

- `Channel()`: default constructor, equivalent to `Channel{Any}(0)`
- `Channel(Inf)`: equivalent to `Channel{Any}(typemax(Int))`
- `Channel(sz)`: equivalent to `Channel{Any}(sz)`

Julia 1.3

The default constructor `Channel()` and default `size=0` were added in Julia 1.3.

source

`Base.Channel` – Method.

```
Channel{T=Any}(func::Function, size=0; taskref=nothing, spawn=false)
```

Create a new task from `func`, bind it to a new channel of type `T` and size `size`, and schedule the task, all in a single call.

`func` must accept the bound channel as its only argument.

If you need a reference to the created task, pass a `Ref{Task}` object via the keyword argument `taskref`.

If `spawn = true`, the Task created for `func` may be scheduled on another thread in parallel, equivalent to creating a task via `Threads.@spawn`.

Return a `Channel`.

Examples

```
julia> chnl = Channel() do ch
           foreach(i -> put!(ch, i), 1:4)
       end;

julia> typeof(chnl)
Channel{Any}

julia> for i in chnl
           @show i
       end;
i = 1
i = 2
i = 3
i = 4
```

Referencing the created task:

```
julia> taskref = Ref{Task}();

julia> chnl = Channel(taskref=taskref) do ch
           println(take!(ch))
       end;

julia> istaskdone(taskref[])
false

julia> put!(chnl, "Hello");
Hello

julia> istaskdone(taskref[])
true
```

Julia 1.3

The `spawn=` parameter was added in Julia 1.3. This constructor was added in Julia 1.3. In earlier versions of Julia, Channel used keyword arguments to set `size` and `T`, but those constructors are deprecated.

```
julia> chnl = Channel{Char}(1, spawn=true) do ch
           for c in "hello world"
               put!(ch, c)
           end
       end
Channel{Char}(sz_max:1,sz_curr:1)

julia> String(collect(chnl))
"hello world"
```

source

`Base.put!` – Method.

```
put!(c::Channel, v)
```

Append an item `v` to the channel `c`. Blocks if the channel is full.

For unbuffered channels, blocks until a `take!` is performed by a different task.

Julia 1.1

v now gets converted to the channel's type with `convert` as `put!` is called.

source

`Base.take!` – Method.

```
take!(c::Channel)
```

Remove and return a value from a `Channel`. Blocks until data is available.

For unbuffered channels, blocks until a `put!` is performed by a different task.

source

`Base.isready` – Method.

```
isready(c::Channel)
```

Determine whether a `Channel` has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns `true` if there are tasks waiting on a `put!`.

source

`Base.fetch` – Method.

```
fetch(c::Channel)
```

Wait for and get the first available item from the channel. Does not remove the item. `fetch` is unsupported on an unbuffered (0-size) channel.

source

`Base.close` – Method.

```
close(c::Channel[, excp::Exception])
```

Close a channel. An exception (optionally given by `excp`), is thrown by:

- `put!` on a closed channel.
- `take!` and `fetch` on an empty, closed channel.

source

`Base.bind` – Method.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of `chnl` with a task. `Channel chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed `Channel` objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1
```

```
julia> put!(c, 1);
ERROR: TaskFailedException:
foo
Stacktrace:
[...]
```

source

Chapter 54

Multi-Threading

`Base.Threads.@threads` – Macro.

```
Threads.@threads [schedule] for ... end
```

A macro to parallelize a `for` loop to run with multiple threads. Splits the iteration space among multiple tasks and runs those tasks on threads according to a scheduling policy. A barrier is placed at the end of the loop which waits for all tasks to finish execution.

The `schedule` argument can be used to request a particular scheduling policy. The only currently supported value is `:static`, which creates one task per thread and divides the iterations equally among them. Specifying `:static` is an error if used from inside another `@threads` loop or from a thread other than 1.

The default schedule (used when no `schedule` argument is present) is subject to change.

> Julia 1.5
>
> The `schedule` argument is available as of Julia 1.5.

source

> Missing docstring.
>
> Missing docstring for `Base.Threads.foreach`. Check Documenter's build log for details.

`Base.Threads.@spawn` – Macro.

```
Threads.@spawn expr
```

Create and run a `Task` on any available thread. To wait for the task to finish, call `wait` on the result of this macro, or call `fetch` to wait and then obtain its return value.

Values can be interpolated into `@spawn` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the value of a variable, isolating the aysnchronous code from changes to the variable's value in the current task.

> Note
>
> See the manual chapter on threading for important caveats.

> Julia 1.3
>
> This macro is available as of Julia 1.3.

> Julia 1.4
>
> Interpolating values via $ is available as of Julia 1.4.

source

`Base.Threads.threadid` – Function.

```
Threads.threadid()
```

Get the ID number of the current thread of execution. The master thread has ID `1`.

source

`Base.Threads.nthreads` – Function.

```
Threads.nthreads()
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `threadid()`.

source

54.1 Synchronization

`Base.Threads.Condition` – Type.

```
Threads.Condition([lock])
```

A thread-safe version of Base.Condition.

To call wait or notify on a Threads.Condition, you must first call lock on it. When wait is called, the lock is atomically released during blocking, and will be reacquired before wait returns. Therefore idiomatic use of a Threads.Condition c looks like the following:

```
lock(c)
try
    while !thing_we_are_waiting_for
        wait(c)
    end
finally
    unlock(c)
end
```

Julia 1.2

This functionality requires at least Julia 1.2.

source

Base.Event – Type.

```
Event()
```

Create a level-triggered event source. Tasks that call wait on an Event are suspended and queued until notify is called on the Event. After notify is called, the Event remains in a signaled state and tasks will no longer block when waiting for it.

Julia 1.1

This functionality requires at least Julia 1.1.

source

See also Synchronization.

54.2 Atomic operations

Warning

The API for atomic operations has not yet been finalized and is likely to change.

Base.Threads.Atomic – Type.

```
Threads.Atomic{T}
```

Holds a reference to an object of type `T`, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive boolean, integer, and float-point types. These are `Bool`, `Int8`...`Int128`, `UInt8`...`UInt128`, and `Float16`...`Float64`.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the `[]` notation:

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> x[] = 1
1

julia> x[]
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

source

Base.Threads.atomic_cas! – Function.

```
Threads.atomic_cas!(x::Atomic{T}, cmp::T, newval::T) where T
```

Atomically compare-and-set `x`

Atomically compares the value in `x` with `cmp`. If equal, write `newval` to `x`. Otherwise, leaves `x` unmodified. Returns the old value in `x`. By comparing the returned value to `cmp` (via `===`) one knows whether `x` was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in `x`. After the transaction, the new value is stored only if `x` has not been modified in the mean time.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

source

Base.Threads.atomic_xchg! – Function.

```
Threads.atomic_xchg!(x::Atomic{T}, newval::T) where T
```

Atomically exchange the value in x

Atomically exchanges the value in x with newval. Returns the old value.

For further details, see LLVM's atomicrmw xchg instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
3

julia> x[]
2
```

source

Base.Threads.atomic_add! – Function.

```
Threads.atomic_add!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically add `val` to `x`

Performs `x[] += val` atomically. Returns the old value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw add` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_add!(x, 2)
3

julia> x[]
5
```

source

`Base.Threads.atomic_sub!` — Function.

```
Threads.atomic_sub!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the old value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw sub` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_sub!(x, 2)
3

julia> x[]
1
```

source

`Base.Threads.atomic_and!` — Function.

```
Threads.atomic_and!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw and` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_and!(x, 2)
3

julia> x[]
2
```

source

Base.Threads.atomic_nand! – Function.

```
Threads.atomic_nand!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-nand (not-and) `x` with `val`

Performs `x[] = ~(x[] & val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw nand` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_nand!(x, 2)
3

julia> x[]
-3
```

source

Base.Threads.atomic_or! – Function.

```
Threads.atomic_or!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-or `x` with `val`

Performs `x[] |= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw or` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_or!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_xor! – Function.

```
Threads.atomic_xor!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-xor (exclusive-or) `x` with `val`

Performs `x[] $= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw xor` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_xor!(x, 7)
5

julia> x[]
2
```

source

Base.Threads.atomic_max! – Function.

```
Threads.atomic_max!(x::Atomic{T}, val::T) where T
```

Atomically store the maximum of `x` and `val` in `x`

Performs `x[] = max(x[], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw max` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_max!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_min! – Function.

```
Threads.atomic_min!(x::Atomic{T}, val::T) where T
```

Atomically store the minimum of `x` and `val` in `x`

Performs `x[] = min(x[], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw min` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(7)
Base.Threads.Atomic{Int64}(7)

julia> Threads.atomic_min!(x, 5)
7

julia> x[]
5
```

source

Base.Threads.atomic_fence – Function.

```
Threads.atomic_fence()
```

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

source

54.3 ccall using a threadpool (Experimental)

Base.@threadcall – Macro.

```
@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as ccall but does the work in a different thread. This is useful when you want to call a blocking C function without causing the main `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

source

54.4 Low-level synchronization primitives

These building blocks are used to create the regular synchronization objects.

Base.Threads.SpinLock – Type.

```
SpinLock()
```

Create a non-reentrant, test-and-test-and-set spin lock. Recursive use will result in a deadlock. This kind of lock should only be used around code that takes little time to execute and does not block (e.g. perform I/O). In general, ReentrantLock should be used instead.

Each `lock` must be matched with an `unlock`.

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, different synchronization approaches should be considered.

source

Chapter 55

Constants

`Core.nothing` – Constant.

```
nothing
```

The singleton instance of type `Nothing`, used by convention when there is no value to return (as in a C `void` function) or when a variable or field holds no value.

source

`Base.PROGRAM_FILE` – Constant.

```
PROGRAM_FILE
```

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see `@__FILE__`.

source

`Base.ARGS` – Constant.

```
ARGS
```

An array of the command line arguments passed to Julia, as strings.

source

`Base.C_NULL` – Constant.

```
C_NULL
```

The C null pointer constant, sometimes used when calling external code.

source

Base.VERSION – Constant.

```
VERSION
```

A `VersionNumber` object describing which version of Julia is in use. For details see Version Number Literals.

source

Base.DEPOT_PATH – Constant.

```
DEPOT_PATH
```

A stack of "depot" locations where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. By default it includes:

1. `~/.julia` where `~` is the user home as appropriate on the system;
2. an architecture-specific shared system directory, e.g. `/usr/local/share/julia`;
3. an architecture-independent shared system directory, e.g. `/usr/share/julia`.

So `DEPOT_PATH` might be:

```
[joinpath(homedir(), ".julia"), "/usr/local/share/julia", "/usr/share/julia"]
```

The first entry is the "user depot" and should be writable by and owned by the current user. The user depot is where: registries are cloned, new package versions are installed, named environments are created and updated, package repos are cloned, newly compiled package image files are saved, log files are written, development packages are checked out by default, and global configuration data is saved. Later entries in the depot path are treated as read-only and are appropriate for registries, packages, etc. installed and managed by system administrators.

`DEPOT_PATH` is populated based on the `JULIA_DEPOT_PATH` environment variable if set.

See also: `JULIA_DEPOT_PATH`, and Code Loading.

source

Base.LOAD_PATH – Constant.

```
LOAD_PATH
```

An array of paths for `using` and `import` statements to consider as project environments or package directories when loading code. It is populated based on the `JULIA_LOAD_PATH` environment variable if set; otherwise it defaults to `["@", "@v#.#", "@stdlib"]`. Entries starting with `@` have special meanings:

- `@` refers to the "current active environment", the initial value of which is initially determined by the `JULIA_PROJECT` environment variable or the `--project` command-line option.
- `@stdlib` expands to the absolute path of the current Julia installation's standard library directory.
- `@name` refers to a named environment, which are stored in depots (see `JULIA_DEPOT_PATH`) under the `environments` subdirectory. The user's named environments are stored in `~/.julia/environments` so `@name` would refer to the environment in `~/.julia/environments/name` if it exists and contains a `Project.toml` file. If `name` contains # characters, then they are replaced with the major, minor and patch components of the Julia version number. For example, if you are running Julia 1.2 then `@v#.#` expands to `@v1.2` and will look for an environment by that name, typically at `~/.julia/environments/v1.2`.

The fully expanded value of `LOAD_PATH` that is searched for projects and packages can be seen by calling the `Base.load_path()` function.

See also: `JULIA_LOAD_PATH`, `JULIA_PROJECT`, `JULIA_DEPOT_PATH`, and Code Loading.

source

`Base.Sys.BINDIR` — Constant.

```
Sys.BINDIR
```

A string containing the full path to the directory containing the `julia` executable.

source

`Base.Sys.CPU_THREADS` — Constant.

```
Sys.CPU_THREADS
```

The number of logical CPU cores available in the system, i.e. the number of threads that the CPU can run concurrently. Note that this is not necessarily the number of CPU cores, for example, in the presence of hyper-threading.

See Hwloc.jl or CpuId.jl for extended information, including number of physical cores.

source

Base.Sys.WORD_SIZE – Constant.

```
Sys.WORD_SIZE
```

Standard word size on the current machine, in bits.

source

Base.Sys.KERNEL – Constant.

```
Sys.KERNEL
```

A symbol representing the name of the operating system, as returned by `uname` of the build configuration.

source

Base.Sys.ARCH – Constant.

```
Sys.ARCH
```

A symbol representing the architecture of the build configuration.

source

Base.Sys.MACHINE – Constant.

```
Sys.MACHINE
```

A string containing the build triple.

source

See also:

- stdin
- stdout
- stderr
- ENV
- ENDIAN_BOM
- Libc.MS_ASYNC
- Libc.MS_INVALIDATE
- Libc.MS_SYNC

Chapter 56

Filesystem

Base.Filesystem.pwd – Function.

```
pwd() -> AbstractString
```

Get the current working directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"
```

source

Base.Filesystem.cd – Method.

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

Examples

```
julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
```

```
"/home/JuliaUser/Projects/julia"

julia> cd()

julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.cd – Method.

```
cd(f::Function, dir::AbstractString=homedir())
```

Temporarily change the current working directory to `dir`, apply function `f` and finally return to the original directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd(readdir, "/home/JuliaUser/Projects/julia")
34-element Array{String,1}:
 ".circleci"
 ".freebsdci.sh"
 ".git"
 ".gitattributes"
 ".github"
 ⋮
 "test"
 "ui"
 "usr"
 "usr-staging"

julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.readdir – Function.

```
readdir(dir::AbstractString=pwd();
    join::Bool = false,
    sort::Bool = true,
) -> Vector{String}
```

Return the names in the directory `dir` or the current working directory if not given. When `join` is false, `readdir` returns just the names in the directory as is; when `join` is true, it returns `joinpath(dir, name)` for each `name` so that the returned strings are full paths. If you want to get absolute paths back, call `readdir` with an absolute directory path and `join` set to true.

By default, `readdir` sorts the list of names it returns. If you want to skip sorting the names and get them in the order that the file system lists them, you can use `readir(dir, sort=false)` to opt out of sorting.

> Julia 1.4
>
> The `join` and `sort` keyword arguments require at least Julia 1.4.

Examples

```
julia> cd("/home/JuliaUser/dev/julia")

julia> readdir()
30-element Array{String,1}:
 ".appveyor.yml"
 ".git"
 ".gitattributes"
 ⋮
 "ui"
 "usr"
 "usr-staging"

julia> readdir(join=true)
30-element Array{String,1}:
 "/home/JuliaUser/dev/julia/.appveyor.yml"
 "/home/JuliaUser/dev/julia/.git"
 "/home/JuliaUser/dev/julia/.gitattributes"
 ⋮
 "/home/JuliaUser/dev/julia/ui"
 "/home/JuliaUser/dev/julia/usr"
 "/home/JuliaUser/dev/julia/usr-staging"
```

```
julia> readdir("base")
145-element Array{String,1}:
 ".gitignore"
 "Base.jl"
 "Enums.jl"
 ⋮
 "version_git.sh"
 "views.jl"
 "weakkeydict.jl"

julia> readdir("base", join=true)
145-element Array{String,1}:
 "base/.gitignore"
 "base/Base.jl"
 "base/Enums.jl"
 ⋮
 "base/version_git.sh"
 "base/views.jl"
 "base/weakkeydict.jl"```

julia> readdir(abspath("base"), join=true)
145-element Array{String,1}:
 "/home/JuliaUser/dev/julia/base/.gitignore"
 "/home/JuliaUser/dev/julia/base/Base.jl"
 "/home/JuliaUser/dev/julia/base/Enums.jl"
 ⋮
 "/home/JuliaUser/dev/julia/base/version_git.sh"
 "/home/JuliaUser/dev/julia/base/views.jl"
 "/home/JuliaUser/dev/julia/base/weakkeydict.jl"
```

source

Base.Filesystem.walkdir – Function.

```
walkdir(dir; topdown=true, follow_symlinks=false, onerror=throw)
```

Return an iterator that walks the directory tree of a directory. The iterator returns a tuple containing `(rootpath, dirs, files)`. The directory tree can be traversed top-down or bottom-up. If `walkdir` encounters a SystemError it will rethrow the error by default. A custom error handling function can be provided through `onerror` keyword argument. `onerror` is called with a `SystemError` as argument.

Examples

```
for (root, dirs, files) in walkdir(".")
    println("Directories in $root")
    for dir in dirs
        println(joinpath(root, dir)) # path to directories
    end
    println("Files in $root")
    for file in files
        println(joinpath(root, file)) # path to files
    end
end
```

```
julia> mkpath("my/test/dir");

julia> itr = walkdir("my");

julia> (root, dirs, files) = first(itr)
("my", ["test"], String[])

julia> (root, dirs, files) = first(itr)
("my/test", ["dir"], String[])

julia> (root, dirs, files) = first(itr)
("my/test/dir", String[], String[])
```

source

Base.Filesystem.mkdir – Function.

```
mkdir(path::AbstractString; mode::Unsigned = 0o777)
```

Make a new directory with name `path` and permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See `mkpath` for a function which creates all required intermediate directories. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"
```

```
julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"
```

source

Base.Filesystem.mkpath – Function.

```
mkpath(path::AbstractString; mode::Unsigned = 0o777)
```

Create all directories in the given `path`, with permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"

julia> mkpath("my/test/dir")
"my/test/dir"

julia> readdir()
1-element Array{String,1}:
 "my"

julia> cd("my")

julia> readdir()
1-element Array{String,1}:
 "test"

julia> readdir("test")
1-element Array{String,1}:
 "dir"
```

source

Base.Filesystem.symlink – Function.

```
symlink(target::AbstractString, link::AbstractString)
```

Creates a symbolic link to `target` with the name `link`.

> Note
>
> This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

source

Base.Filesystem.readlink – Function.

```
readlink(path::AbstractString) -> AbstractString
```

Return the target location a symbolic link `path` points to.

source

Base.Filesystem.chmod – Function.

```
chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. `0o777`) are currently supported. If `recursive=true` and the path is a directory all permissions in that directory will be recursively changed. Return `path`.

source

Base.Filesystem.chown – Function.

```
chown(path::AbstractString, owner::Integer, group::Integer=-1)
```

Change the owner and/or group of `path` to `owner` and/or `group`. If the value entered for `owner` or `group` is `-1` the corresponding ID will not change. Only integer `owner`s and `group`s are currently supported. Return `path`.

source

Base.Libc.RawFD – Type.

```
RawFD
```

Primitive type which wraps the native OS file descriptor. `RawFD`s can be passed to methods like `stat` to discover information about the underlying file, and can also be used to open streams, with the `RawFD` describing the OS file backing the stream.

source

`Base.stat` – Function.

```
stat(file)
```

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated
mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file was created

source

`Base.Filesystem.lstat` – Function.

```
lstat(file)
```

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

source

Base.Filesystem.ctime – Function.

```
ctime(file)
```

Equivalent to `stat(file).ctime`.

source

Base.Filesystem.mtime – Function.

```
mtime(file)
```

Equivalent to `stat(file).mtime`.

source

Base.Filesystem.filemode – Function.

```
filemode(file)
```

Equivalent to `stat(file).mode`.

source

Base.Filesystem.filesize – Function.

```
filesize(path...)
```

Equivalent to `stat(file).size`.

source

Base.Filesystem.uperm – Function.

```
uperm(file)
```

Get the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see stat.

source

Base.Filesystem.gperm – Function.

```
gperm(file)
```

Like uperm but gets the permissions of the group owning the file.

source

Base.Filesystem.operm – Function.

```
operm(file)
```

Like uperm but gets the permissions for people who neither own the file nor are a member of the group owning the file

source

Base.Filesystem.cp – Function.

```
cp(src::AbstractString, dst::AbstractString; force::Bool=false, follow_symlinks::Bool=false)
```

Copy the file, link, or directory from src to dst. force=true will first remove an existing dst.

If follow_symlinks=false, and src is a symbolic link, dst will be created as a symbolic link. If follow_symlinks=true and src is a symbolic link, dst will be a copy of the file or directory src refers to. Return dst.

source

Base.download – Function.

```
download(url::AbstractString, [localfile::AbstractString])
```

Download a file from the given url, optionally renaming it to the given local file name. If no filename is given this will download into a randomly-named file in your temp directory. Note that this function relies on the availability of external tools such as curl, wget or fetch to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

Returns the filename of the downloaded file.

source

Base.Filesystem.mv – Function.

```
mv(src::AbstractString, dst::AbstractString; force::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`. Return `dst`.

Examples

```
julia> write("hello.txt", "world");

julia> mv("hello.txt", "goodbye.txt")
"goodbye.txt"

julia> "hello.txt" in readdir()
false

julia> readline("goodbye.txt")
"world"

julia> write("hello.txt", "world2");

julia> mv("hello.txt", "goodbye.txt")
ERROR: ArgumentError: 'goodbye.txt' exists. `force=true` is required to remove 'goodbye.txt' before moving.
Stacktrace:
 [1] #checkfor_mv_cp_cptree#10(::Bool, ::Function, ::String, ::String, ::String) at ./file.jl:293
[...]

julia> mv("hello.txt", "goodbye.txt", force=true)
"goodbye.txt"

julia> rm("goodbye.txt");
```

source

Base.Filesystem.rm – Function.

```
rm(path::AbstractString; force::Bool=false, recursive::Bool=false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

Examples

```
julia> mkpath("my/test/dir");

julia> rm("my", recursive=true)

julia> rm("this_file_does_not_exist", force=true)

julia> rm("this_file_does_not_exist")
ERROR: IOError: unlink: no such file or directory (ENOENT)
Stacktrace:
[...]
```

source

Base.Filesystem.touch – Function.

```
touch(path::AbstractString)
```

Update the last-modified timestamp on a file to the current time.

If the file does not exist a new file is created.

Return `path`.

Examples

```
julia> write("my_little_file", 2);

julia> mtime("my_little_file")
1.5273815391135583e9

julia> touch("my_little_file");

julia> mtime("my_little_file")
1.527381559163435e9
```

We can see the `mtime` has been modified by `touch`.

source

Base.Filesystem.tempname – Function.

```
tempname(parent=tempdir(); cleanup=true) -> String
```

Generate a temporary file path. This function only returns a path; no file is created. The path is likely to be unique, but this cannot be guaranteed due to the very remote posibility of two simultaneous calls to `tempname` generating the same file name. The name is guaranteed to differ from all files already existing at the time of the call to `tempname`.

When called with no arguments, the temporary name will be an absolute path to a temporary name in the system temporary directory as given by `tempdir()`. If a `parent` directory argument is given, the temporary path will be in that directory instead.

The `cleanup` option controls whether the process attempts to delete the returned path automatically when the process exits. Note that the `tempname` function does not create any file or directory at the returned location, so there is nothing to cleanup unless you create a file or directory there. If you do and `clean` is `true` it will be deleted upon process termination.

> Julia 1.4
>
> The `parent` and `cleanup` arguments were added in 1.4. Prior to Julia 1.4 the path `tempname` would never be cleaned up at process termination.

> Warning
>
> This can lead to security holes if another process obtains the same file name and creates the file before you are able to. Open the file with `JL_O_EXCL` if this is a concern. Using `mktemp()` is also recommended instead.

source

`Base.Filesystem.tempdir` — Function.

```
tempdir()
```

Gets the path of the temporary directory. On Windows, `tempdir()` uses the first environment variable found in the ordered list `TMP`, `TEMP`, `USERPROFILE`. On all other operating systems, `tempdir()` uses the first environment variable found in the ordered list `TMPDIR`, `TMP`, `TEMP`, and `TEMPDIR`. If none of these are found, the path `"/tmp"` is used.

source

`Base.Filesystem.mktemp` — Method.

```
mktemp(parent=tempdir(); cleanup=true) -> (path, io)
```

Return `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path. The `cleanup` option controls whether the temporary file is automatically deleted when the process exits.

source

Missing docstring.

Missing docstring for `Base.Filesystem.mktemp(::Function, ::Any)`. Check Documenter's build log for details.

Missing docstring.

Missing docstring for `Base.Filesystem.mktempdir(::Any)`. Check Documenter's build log for details.

Missing docstring.

Missing docstring for `Base.Filesystem.mktempdir(::Function, ::Any)`. Check Documenter's build log for details.

`Base.Filesystem.isblockdev` — Function.

```
isblockdev(path) -> Bool
```

Return `true` if `path` is a block device, `false` otherwise.

source

`Base.Filesystem.ischardev` — Function.

```
ischardev(path) -> Bool
```

Return `true` if `path` is a character device, `false` otherwise.

source

`Base.Filesystem.isdir` — Function.

```
isdir(path) -> Bool
```

Return `true` if `path` is a directory, `false` otherwise.

Examples

```
julia> isdir(homedir())
true

julia> isdir("not/a/directory")
false
```

See also: `isfile` and `ispath`.

source

`Base.Filesystem.isfifo` – Function.

```
isfifo(path) -> Bool
```

Return `true` if `path` is a FIFO, `false` otherwise.

source

`Base.Filesystem.isfile` – Function.

```
isfile(path) -> Bool
```

Return `true` if `path` is a regular file, `false` otherwise.

Examples

```
julia> isfile(homedir())
false

julia> f = open("test_file.txt", "w");

julia> isfile(f)
true

julia> close(f); rm("test_file.txt")
```

See also: `isdir` and `ispath`.

source

`Base.Filesystem.islink` – Function.

```
islink(path) -> Bool
```

Return `true` if `path` is a symbolic link, `false` otherwise.

source

`Base.Filesystem.ismount` — Function.

```
ismount(path) -> Bool
```

Return `true` if `path` is a mount point, `false` otherwise.

source

`Base.Filesystem.ispath` — Function.

```
ispath(path) -> Bool
```

Return `true` if a valid filesystem entity exists at `path`, otherwise returns `false`. This is the generalization of `isfile`, `isdir` etc.

source

`Base.Filesystem.issetgid` — Function.

```
issetgid(path) -> Bool
```

Return `true` if `path` has the setgid flag set, `false` otherwise.

source

`Base.Filesystem.issetuid` — Function.

```
issetuid(path) -> Bool
```

Return `true` if `path` has the setuid flag set, `false` otherwise.

source

`Base.Filesystem.issocket` — Function.

```
issocket(path) -> Bool
```

Return `true` if `path` is a socket, `false` otherwise.

source

Base.Filesystem.issticky – Function.

```
issticky(path) -> Bool
```

Return `true` if `path` has the sticky bit set, `false` otherwise.

source

Base.Filesystem.homedir – Function.

```
homedir() -> String
```

Return the current user's home directory.

> Note
>
> `homedir` determines the home directory via `libuv`'s `uv_os_homedir`. For details (for example on how to specify the home directory via environment variables), see the `uv_os_homedir` documentation.

source

Base.Filesystem.dirname – Function.

```
dirname(path::AbstractString) -> AbstractString
```

Get the directory part of a path. Trailing characters ('/' or '₩') in the path are counted as part of the path.

Examples

```
julia> dirname("/home/myuser")
"/home"

julia> dirname("/home/myuser/")
"/home/myuser"
```

See also: basename

source

Base.Filesystem.basename – Function.

```
basename(path::AbstractString) -> AbstractString
```

Get the file name part of a path.

Examples

```
julia> basename("/home/myuser/example.jl")
"example.jl"
```

See also: dirname

source

Base.@__FILE__ – Macro.

```
@__FILE__ -> AbstractString
```

Expand to a string with the path to the file containing the macrocall, or an empty string if evaluated by `julia -e <expr>`. Return `nothing` if the macro was missing parser source information. Alternatively see PROGRAM_FILE.

source

Base.@__DIR__ – Macro.

```
@__DIR__ -> AbstractString
```

Expand to a string with the absolute path to the directory of the file containing the macrocall. Return the current working directory if run from a REPL or if evaluated by `julia -e <expr>`.

source

Base.@__LINE__ – Macro.

```
@__LINE__ -> Int
```

Expand to the line number of the location of the macrocall. Return `0` if the line number could not be determined.

source

Base.Filesystem.isabspath – Function.

```
isabspath(path::AbstractString) -> Bool
```

Determine whether a path is absolute (begins at the root directory).

Examples

```
julia> isabspath("/home")
true

julia> isabspath("home")
false
```

source

Base.Filesystem.isdirpath – Function.

```
isdirpath(path::AbstractString) -> Bool
```

Determine whether a path refers to a directory (for example, ends with a path separator).

Examples

```
julia> isdirpath("/home")
false

julia> isdirpath("/home/")
true
```

source

Base.Filesystem.joinpath – Function.

```
joinpath(parts::AbstractString...) -> String
```

Join path components into a full path. If some argument is an absolute path or (on Windows) has a drive specification that doesn't match the drive computed for the join of the preceding paths, then prior components are dropped.

Note on Windows since there is a current directory for each drive, `joinpath("c:", "foo")` represents a path relative to the current directory on drive "c:" so this is equal to "c:foo", not "c:₩foo". Furthermore, `joinpath` treats this as a non-absolute path and ignores the drive letter casing, hence `joinpath("C:\A","c:b") = "C:\A\b"`.

Examples

```
julia> joinpath("/home/myuser", "example.jl")
"/home/myuser/example.jl"
```

source

Base.Filesystem.abspath – Function.

```
abspath(path::AbstractString) -> String
```

Convert a path to an absolute path by adding the current directory if necessary. Also normalizes the path as in normpath.

source

```
abspath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to abspath(joinpath(path, paths...)).

source

Base.Filesystem.normpath – Function.

```
normpath(path::AbstractString) -> String
```

Normalize a path, removing "." and ".." entries.

Examples

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"
```

source

```
normpath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to a normalized path by joining them together and removing "." and ".." entries. Equivalent to normpath(joinpath(path, paths...)).

source

Base.Filesystem.realpath – Function.

```
realpath(path::AbstractString) -> String
```

Canonicalize a path by expanding symbolic links and removing "." and ".." entries. On case-insensitive case-preserving filesystems (typically Mac and Windows), the filesystem's stored case for the path is returned.

(This function throws an exception if path does not exist in the filesystem.)

source

Base.Filesystem.relpath – Function.

```
relpath(path::AbstractString, startpath::AbstractString = ".") -> AbstractString
```

Return a relative filepath to `path` either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `startpath`.

source

Base.Filesystem.expanduser – Function.

```
expanduser(path::AbstractString) -> AbstractString
```

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

source

Base.Filesystem.splitdir – Function.

```
splitdir(path::AbstractString) -> (AbstractString, AbstractString)
```

Split a path into a tuple of the directory name and file name.

Examples

```
julia> splitdir("/home/myuser")
("/home", "myuser")
```

source

Base.Filesystem.splitdrive – Function.

```
splitdrive(path::AbstractString) -> (AbstractString, AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

source

Base.Filesystem.splitext – Function.

```
splitext(path::AbstractString) -> (AbstractString, AbstractString)
```

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

Examples

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example")
("/home/myuser/example", "")
```

source

Base.Filesystem.splitpath – Function.

```
splitpath(path::AbstractString) -> Vector{String}
```

Split a file path into all its path components. This is the opposite of `joinpath`. Returns an array of substrings, one for each directory or file in the path, including the root directory if present.

> Julia 1.1
>
> This function requires at least Julia 1.1.

Examples

```
julia> splitpath("/home/myuser/example.jl")
4-element Array{String,1}:
 "/"
 "home"
 "myuser"
 "example.jl"
```

source

Chapter 57

I/O and Network

57.1 General I/O

`Base.stdout` — Constant.

```
stdout
```

Global variable referring to the standard out stream.

source

`Base.stderr` — Constant.

```
stderr
```

Global variable referring to the standard error stream.

source

`Base.stdin` — Constant.

```
stdin
```

Global variable referring to the standard input stream.

source

`Base.open` — Function.

```
open(f::Function, args...; kwargs...)
```

Apply the function `f` to the result of `open(args...; kwargs...)` and close the resulting file descriptor upon completion.

Examples

```
julia> open("myfile.txt", "w") do io
           write(io, "Hello world!")
       end;

julia> open(f->read(f, String), "myfile.txt")
"Hello world!"

julia> rm("myfile.txt")
```

source

```
open(filename::AbstractString; lock = true, keywords...) -> IOStream
```

Open a file in a mode specified by five boolean keyword arguments:

Keyword	Description	Default
`read`	open for reading	`!write`
`write`	open for writing	`truncate | append`
`create`	create if non-existent	`!read & write | truncate | append`
`truncate`	truncate to zero size	`!read & write`
`append`	seek to end	`false`

The default when no keywords are passed is to open files for reading only. Returns a stream for accessing the opened file.

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

> Julia 1.5
>
> The `lock` argument is available as of Julia 1.5.

source

```
open(filename::AbstractString, [mode::AbstractString]; lock = true) -> IOStream
```

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

Mode	Description	Keywords
r	read	none
w	write, create, truncate	write = true
a	write, create, append	append = true
r+	read, write	read = true, write = true
w+	read, write, create, truncate	truncate = true, read = true
a+	read, write, create, append	append = true, read = true

The lock keyword argument controls whether operations will be locked for safe multi-threaded access.

Examples

```
julia> io = open("myfile.txt", "w");

julia> write(io, "Hello world!");

julia> close(io);

julia> io = open("myfile.txt", "r");

julia> read(io, String)
"Hello world!"

julia> write(io, "This file is read only")
ERROR: ArgumentError: write failed, IOStream is not writeable
[...]

julia> close(io)

julia> io = open("myfile.txt", "a");

julia> write(io, "This stream is not read only")
28

julia> close(io)

julia> rm("myfile.txt")
```

Julia 1.5

The `lock` argument is available as of Julia 1.5.

source

```
open(fd::OS_HANDLE) -> IO
```

Take a raw file descriptor wrap it in a Julia-aware IO type, and take ownership of the fd handle. Call `open(Libc.dup(fd))` to avoid the ownership capture of the original handle.

Warn

Do not call this on a handle that's already owned by some other part of the system.

source

```
open(command, mode::AbstractString, stdio=devnull)
```

Run `command` asynchronously. Like `open(command, stdio; read, write)` except specifying the read and write flags via a mode string instead of keyword arguments. Possible mode strings are:

Mode	Description	Keywords
`r`	read	none
`w`	write	`write = true`
`r+`	read, write	`read = true, write = true`
`w+`	read, write	`read = true, write = true`

source

```
open(command, stdio=devnull; write::Bool = false, read::Bool = !write)
```

Start running `command` asynchronously, and return a `process::IO` object. If `read` is true, then reads from the process come from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `write` is true, then writes go to the process's standard input and `stdio` optionally specifies the process's standard output stream. The process's standard error stream is connected to the current global `stderr`.

source

```
open(f::Function, command, args...; kwargs...)
```

Similar to `open(command, args...; kwargs...)`, but calls `f(stream)` on the resulting process stream, then closes the input stream and waits for the process to complete. Returns the value returned by `f`.

source

Base.IOStream – Type.

```
IOStream
```

A buffered IO stream wrapping an OS file descriptor. Mostly used to represent files returned by open.

source

Base.IOBuffer – Type.

```
IOBuffer([data::AbstractVector{UInt8}]; keywords...) -> IOBuffer
```

Create an in-memory I/O stream, which may optionally operate on a pre-existing array.

It may take optional keyword arguments:

- read, write, append: restricts operations to the buffer; see open for details.
- truncate: truncates the buffer size to zero length.
- maxsize: specifies a size beyond which the buffer may not be grown.
- sizehint: suggests a capacity of the buffer (data must implement sizehint!(data, size)).

When data is not given, the buffer will be both readable and writable by default.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> io = IOBuffer(b"JuliaLang is a GitHub organization.")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=35, maxsize=Inf,
↪  ptr=1, mark=-1)

julia> read(io, String)
"JuliaLang is a GitHub organization."

julia> write(io, "This isn't writable.")
```

```
ERROR: ArgumentError: ensureroom failed, IOBuffer is not writeable

julia> io = IOBuffer(UInt8[], read=true, write=true, maxsize=34)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0, maxsize=34, ptr=1,
↪  mark=-1)

julia> write(io, "JuliaLang is a GitHub organization.")
34

julia> String(take!(io))
"JuliaLang is a GitHub organization"

julia> length(read(IOBuffer(b"data", read=true, truncate=false)))
4

julia> length(read(IOBuffer(b"data", read=true, truncate=true)))
0
```

source

```
IOBuffer(string::String)
```

Create a read-only `IOBuffer` on the data underlying the given string.

Examples

```
julia> io = IOBuffer("Haho");

julia> String(take!(io))
"Haho"

julia> String(take!(io))
"Haho"
```

source

`Base.take!` – Method.

```
take!(b::IOBuffer)
```

Obtain the contents of an `IOBuffer` as an array, without copying. Afterwards, the `IOBuffer` is reset to its initial state.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."
```

source

Base.fdio – Function.

```
fdio([name::AbstractString, ]fd::Integer[, own::Bool=false]) -> IOStream
```

Create an IOStream object from an integer file descriptor. If own is true, closing this object will close the underlying descriptor. By default, an IOStream is closed when it is garbage collected. name allows you to associate the descriptor with a named file.

source

Base.flush – Function.

```
flush(stream)
```

Commit all currently buffered writes to the given stream.

source

Base.close – Function.

```
close(stream)
```

Close an I/O stream. Performs a flush first.

source

Base.write – Function.

```
write(io::IO, x)
write(filename::AbstractString, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Return the number of bytes written into the stream. See also `print` to write a text representation (with an encoding that may depend upon `io`).

The endianness of the written value depends on the endianness of the host system. Convert to/from a fixed endianness when writing/reading (e.g. using `htol` and `ltoh`) to get results that are consistent across platforms.

You can write multiple values with the same `write` call. i.e. the following are equivalent:

```
write(io, x, y...)
write(io, x) + write(io, y...)
```

Examples

Consistent serialization:

```
julia> fname = tempname(); # random temporary filename

julia> open(fname,"w") do f
           # Make sure we write 64bit integer in little-endian byte order
           write(f,htol(Int64(42)))
       end
8

julia> open(fname,"r") do f
           # Convert back to host byte order and host integer type
           Int(ltoh(read(f,Int64)))
       end
42
```

Merging write calls:

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> write(io, "Sometimes those members") + write(io, " write documentation.")
44
```

```
julia> String(take!(io))
"Sometimes those members write documentation."
```

User-defined plain-data types without `write` methods can be written when wrapped in a `Ref`:

```
julia> struct MyStruct; x::Float64; end

julia> io = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0, maxsize=Inf,
↪  ptr=1, mark=-1)

julia> write(io, Ref(MyStruct(42.0)))
8

julia> seekstart(io); read!(io, Ref(MyStruct(NaN)))
Base.RefValue{MyStruct}(MyStruct(42.0))
```

source

`Base.read` – Function.

```
read(io::IO, T)
```

Read a single value of type `T` from `io`, in canonical binary representation.

Note that Julia does not convert the endianness for you. Use `ntoh` or `ltoh` for this purpose.

```
read(io::IO, String)
```

Read the entirety of `io`, as a `String`.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, String)
"JuliaLang is a GitHub organization"
```

source

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

source

```
read(s::IO, nb=typemax(Int))
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

source

```
read(s::IOStream, nb::Integer; all=true)
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

```
read(command::Cmd)
```

Run `command` and return the resulting output as an array of bytes.

source

```
read(command::Cmd, String)
```

Run `command` and return the resulting output as a `String`.

source

`Base.read!` – Function.

```
read!(stream::IO, array::AbstractArray)
read!(filename::AbstractString, array::AbstractArray)
```

Read binary data from an I/O stream or file, filling in `array`.

source

Base.readbytes! – Function.

```
readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

source

```
readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=length(b); all::Bool=true)
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

Base.unsafe_read – Function.

```
unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from the `IO` stream object into `ref` (converted to a pointer).

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_read(s::T, p::Ptr{UInt8}, n::UInt)`

source

Base.unsafe_write – Function.

```
unsafe_write(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from `ref` (converted to a pointer) into the `IO` object.

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_write(s::T, p::Ptr{UInt8}, n::UInt)`

source

`Base.position` – Function.

```
position(s)
```

Get the current position of a stream.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> position(io)
5

julia> skip(io, 10);

julia> position(io)
15

julia> seekend(io);

julia> position(io)
35
```

source

`Base.seek` – Function.

```
seek(s, pos)
```

Seek a stream to the given position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)
```

source

Base.seekstart – Function.

```
seekstart(s)
```

Seek a stream to its beginning.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)

julia> seekstart(io);

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
```

source

Base.seekend – Function.

```
seekend(s)
```

Seek a stream to its end.

source

Base.skip – Function.

```
skip(s, offset)
```

Seek a stream relative to the current position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);
```

```
julia> skip(io, 10);

julia> read(io, Char)
'G': ASCII/Unicode U+0047 (category Lu: Letter, uppercase)
```

source

Base.mark – Function.

```
mark(s)
```

Add a mark at the current position of stream `s`. Return the marked position.

See also unmark, reset, ismarked.

source

Base.unmark – Function.

```
unmark(s)
```

Remove a mark from stream `s`. Return `true` if the stream was marked, `false` otherwise.

See also mark, reset, ismarked.

source

Base.reset – Function.

```
reset(s)
```

Reset a stream `s` to a previously marked position, and remove the mark. Return the previously marked position. Throw an error if the stream is not marked.

See also mark, unmark, ismarked.

source

Base.ismarked – Function.

```
ismarked(s)
```

Return `true` if stream `s` is marked.

See also `mark`, `unmark`, `reset`.

source

`Base.eof` — Function.

```
eof(stream) -> Bool
```

Test whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`. `eof` will return `false` as long as buffered data is still available, even if the remote end of a connection is closed.

source

`Base.isreadonly` — Function.

```
isreadonly(io) -> Bool
```

Determine whether a stream is read-only.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> isreadonly(io)
true

julia> io = IOBuffer();

julia> isreadonly(io)
false
```

source

`Base.iswritable` — Function.

```
iswritable(io) -> Bool
```

Return `true` if the specified IO object is writable (if that can be determined).

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           iswritable(io)
       end
true

julia> open("myfile.txt", "r") do io
           iswritable(io)
       end
false

julia> rm("myfile.txt")
```

source

Base.isreadable – Function.

```
isreadable(io) -> Bool
```

Return `true` if the specified IO object is readable (if that can be determined).

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           isreadable(io)
       end
false

julia> open("myfile.txt", "r") do io
           isreadable(io)
       end
true

julia> rm("myfile.txt")
```

source

Base.isopen – Function.

```
isopen(object) -> Bool
```

Determine whether an object - such as a stream or timer – is not yet closed. Once an object is closed, it will never produce a new event. However, since a closed stream may still have data to read in its buffer, use `eof` to check for the ability to read data. Use the `FileWatching` package to be notified when a stream might be writable or readable.

Examples

```
julia> io = open("my_file.txt", "w+");

julia> isopen(io)
true

julia> close(io)

julia> isopen(io)
false
```

source

`Base.Grisu.print_shortest` – Function.

```
print_shortest(io::IO, x)
```

Print the shortest possible representation, with the minimum number of consecutive non-zero digits, of number `x`, ensuring that it would parse to the exact same number.

source

`Base.fd` – Function.

```
fd(stream)
```

Return the file descriptor backing the stream or file. Note that this function only applies to synchronous `File`'s and `IOStream`'s not to any of the asynchronous streams.

source

`Base.redirect_stdout` – Function.

```
redirect_stdout([stream]) -> (rd, wr)
```

Create a pipe to which all C and Julia level `stdout` output will be redirected. Returns a tuple `(rd, wr)` representing the pipe ends. Data written to `stdout` may now be read from the `rd` end of the pipe. The `wr` end is given for convenience in case the old `stdout` object was cached by the user and needs to be replaced elsewhere.

If called with the optional `stream` argument, then returns `stream` itself.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

source

`Base.redirect_stdout` – Method.

```
redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `stdout` to `stream`. Upon completion, `stdout` is restored to its prior setting.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

source

`Base.redirect_stderr` – Function.

```
redirect_stderr([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `stderr`.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

source

`Base.redirect_stderr` – Method.

```
redirect_stderr(f::Function, stream)
```

Run the function `f` while redirecting `stderr` to `stream`. Upon completion, `stderr` is restored to its prior setting.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

source

Base.redirect_stdin – Function.

```
redirect_stdin([stream]) -> (rd, wr)
```

Like redirect_stdout, but for stdin. Note that the order of the return tuple is still (rd, wr), i.e. data to be read from stdin may be written to wr.

Note

stream must be a TTY, a Pipe, or a socket.

source

Base.redirect_stdin – Method.

```
redirect_stdin(f::Function, stream)
```

Run the function f while redirecting stdin to stream. Upon completion, stdin is restored to its prior setting.

Note

stream must be a TTY, a Pipe, or a socket.

source

Base.readchomp – Function.

```
readchomp(x)
```

Read the entirety of x as a string and remove a single trailing newline if there is one. Equivalent to chomp(read(x, String)).

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end;

julia> readchomp("my_file.txt")
"JuliaLang is a GitHub organization.\nIt has many members."

julia> rm("my_file.txt");
```

source

Base.truncate – Function.

```
truncate(file, n)
```

Resize the file or buffer given by the first argument to exactly `n` bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.")
35

julia> truncate(io, 15)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=15, maxsize=Inf,
↪  ptr=16, mark=-1)

julia> String(take!(io))
"JuliaLang is a "

julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.");

julia> truncate(io, 40);

julia> String(take!(io))
"JuliaLang is a GitHub organization.\0\0\0\0\0"
```

source

Base.skipchars – Function.

```
skipchars(predicate, io::IO; linecomment=nothing)
```

Advance the stream `io` such that the next-read character will be the first remaining for which `predicate` returns `false`. If the keyword argument `linecomment` is specified, all characters from that character until the start of the next line are ignored.

Examples

```
julia> buf = IOBuffer("    text")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8, maxsize=Inf,
↪  ptr=1, mark=-1)

julia> skipchars(isspace, buf)
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8, maxsize=Inf,
↪  ptr=5, mark=-1)

julia> String(readavailable(buf))
"text"
```

source

Base.countlines – Function.

```
countlines(io::IO; eol::AbstractChar = '\n')
```

Read `io` until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than `'\n'` are supported by passing them as the second argument. The last non-empty line of `io` is counted even if it does not end with the EOL, matching the length returned by eachline and readlines.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n");

julia> countlines(io)
1

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io)
1

julia> countlines(io, eol = '.')
0
```

source

Base.PipeBuffer – Function.

```
PipeBuffer(data::Vector{UInt8}=UInt8[]; maxsize::Integer = typemax(Int))
```

An IOBuffer that allows reading and performs writes by appending. Seeking and truncating are not supported. See IOBuffer for the available constructors. If data is given, creates a PipeBuffer to operate on a data vector, optionally specifying a size beyond which the underlying Array may not be grown.

source

Base.readavailable – Function.

```
readavailable(stream)
```

Read all available data on the stream, blocking the task only if no data is available. The result is a Vector{UInt8}.

source

Base.IOContext – Type.

```
IOContext
```

IOContext provides a mechanism for passing output configuration settings among show methods.

In short, it is an immutable dictionary that is a subclass of IO. It supports standard dictionary operations such as getindex, and can also be used as an I/O stream.

source

Base.IOContext – Method.

```
IOContext(io::IO, KV::Pair...)
```

Create an IOContext that wraps a given stream, adding the specified key=>value pairs to the properties of that stream (note that io can itself be an IOContext).

- use (key => value) in io to see if this particular combination is in the properties set
- use get(io, key, default) to retrieve the most recent value for a particular key

The following properties are in common use:

- :compact: Boolean specifying that values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements. :compact output should not contain line breaks.
- :limit: Boolean specifying that containers should be truncated, e.g. showing … in place of most elements.

- `:displaysize`: A `Tuple{Int,Int}` giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the `displaysize` function.
- `:typeinfo`: a `Type` characterizing the information already printed concerning the type of the object about to be displayed. This is mainly useful when displaying a collection of objects of the same type, so that redundant type information can be avoided (e.g. `[Float16(0)]` can be shown as "Float16[0.0]" instead of "Float16[Float16(0.0)]" : while displaying the elements of the array, the `:typeinfo` property will be set to `Float16`).
- `:color`: Boolean specifying whether ANSI color/escape codes are supported/expected. By default, this is determined by whether `io` is a compatible terminal and by any `--color` command-line flag when `julia` was launched.

Examples

```
julia> io = IOBuffer();

julia> printstyled(IOContext(io, :color => true), "string", color=:red)

julia> String(take!(io))
"\e[31mstring\e[39m"

julia> printstyled(io, "string", color=:red)

julia> String(take!(io))
"string"
```

```
julia> print(IOContext(stdout, :compact => false), 1.12341234)
1.12341234
julia> print(IOContext(stdout, :compact => true), 1.12341234)
1.12341
```

```
julia> function f(io::IO)
           if get(io, :short, false)
               print(io, "short")
           else
               print(io, "looooong")
           end
       end
f (generic function with 1 method)
```

```
julia> f(stdout)
looooong
julia> f(IOContext(stdout, :short => true))
short
```

source

Base.IOContext – Method.

```
IOContext(io::IO, context::IOContext)
```

Create an `IOContext` that wraps an alternate `IO` but inherits the properties of `context`.

source

57.2 Text I/O

Missing docstring.

Missing docstring for `Base.show(::Any)`. Check Documenter's build log for details.

Base.summary – Function.

```
summary(io::IO, x)
str = summary(x)
```

Print to a stream `io`, or return a string `str`, giving a brief description of a value. By default returns `string(typeof(x))`, e.g. Int64.

For arrays, returns a string of size and type info, e.g. `10-element Array{Int64,1}`.

Examples

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Array{Float64,1}"
```

source

Base.print – Function.

```
print([io::IO], xs...)
```

Write to `io` (or to the default output stream `stdout` if `io` is not given) a canonical (un-decorated) text representation. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

`print` falls back to calling `show`, so most types should just define `show`. Define `print` if your type has a separate "plain" representation. For example, `show` displays strings with quotes, and `print` displays strings without quotes.

`string` returns the output of `print` as a string.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello", ' ', :World!)

julia> String(take!(io))
"Hello World!"
```

source

Base.println – Function.

```
println([io::IO], xs...)
```

Print (using `print`) `xs` followed by a newline. If `io` is not supplied, prints to `stdout`.

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello, world")

julia> String(take!(io))
"Hello, world\n"
```

source

Base.printstyled – Function.

```
printstyled([io], xs...; bold::Bool=false, color::Union{Symbol,Int}=:normal)
```

Print `xs` in a color specified as a symbol or integer, optionally in bold.

`color` may take any of the values `:normal`, `:default`, `:bold`, `:black`, `:blink`, `:blue`, `:cyan`, `:green`, `:hidden`, `:light_black`, `:light_blue`, `:light_cyan`, `:light_green`, `:light_magenta`, `:light_red`, `:light_yellow`, `:magenta`, `:nothing`, `:red`, `:reverse`, `:underline`, `:white`, or `:yellow` or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword `bold` is given as `true`, the result will be printed in bold.

source

Base.sprint – Function.

```
sprint(f::Function, args...; context=nothing, sizehint=0)
```

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string. `context` can be either an IOContext whose properties will be used, or a `Pair` specifying a property and its value. `sizehint` suggests the capacity of the buffer (in bytes).

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or IOContext object whose attributes are used for the I/O stream passed to `f`. The optional `sizehint` is a suggested size (in bytes) to allocate for the buffer used to write the string.

Examples

```
julia> sprint(show, 66.66666; context=:compact => true)
"66.6667"


julia> sprint(showerror, BoundsError([1], 100))
"BoundsError: attempt to access 1-element Array{Int64,1} at index [100]"
```

source

Base.showerror – Function.

```
showerror(io, e)
```

Show a descriptive representation of an exception object `e`. This method is used to display the exception after a call to throw.

Examples

```
julia> struct MyException <: Exception
           msg::AbstractString
       end

julia> function Base.showerror(io::IO, err::MyException)
           print(io, "MyException: ")
           print(io, err.msg)
       end

julia> err = MyException("test exception")
MyException("test exception")

julia> sprint(showerror, err)
"MyException: test exception"

julia> throw(MyException("test exception"))
ERROR: MyException: test exception
```

source

Base.dump – Function.

```
dump(x; maxdepth=8)
```

Show every part of the representation of a value. The depth of the output is truncated at `maxdepth`.

Examples

```
julia> struct MyStruct
           x
           y
       end

julia> x = MyStruct(1, (2,3));

julia> dump(x)
MyStruct
  x: Int64 1
  y: Tuple{Int64,Int64}
    1: Int64 2
    2: Int64 3
```

```
julia> dump(x; maxdepth = 1)
MyStruct
  x: Int64 1
  y: Tuple{Int64,Int64}
```

source

Base.Meta.@dump – Macro.

```
@dump expr
```

Show every part of the representation of the given expression. Equivalent to dump(:(expr)).

source

Base.readline – Function.

```
readline(io::IO=stdin; keep::Bool=false)
readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to stdin). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '\n' or "\r\n" or the end of an input stream. When keep is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When keep is true, they are returned as part of the line.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
```

source

`Base.readuntil` – Function.

```
readuntil(stream::IO, delim; keep::Bool = false)
readuntil(filename::AbstractString, delim; keep::Bool = false)
```

Read a string from an I/O stream or a file, up to the given delimiter. The delimiter can be a `UInt8`, `AbstractChar`, string, or vector. Keyword argument `keep` controls whether the delimiter is included in the result. The text is assumed to be encoded in UTF-8.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57

julia> readuntil("my_file.txt", 'L')
"Julia"

julia> readuntil("my_file.txt", '.', keep = true)
"JuliaLang is a GitHub organization."

julia> rm("my_file.txt")
```

source

`Base.readlines` – Function.

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57
```

```
julia> readlines("my_file.txt")
2-element Array{String,1}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Array{String,1}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

source

Base.eachline – Function.

```
eachline(io::IO=stdin; keep::Bool=false)
eachline(filename::AbstractString; keep::Bool=false)
```

Create an iterable `EachLine` object that will yield each line from an I/O stream or a file. Iteration calls `readline` on the stream argument repeatedly with `keep` passed through, determining whether trailing end-of-line characters are retained. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the `EachLine` object is garbage collected.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\n It has many members.\n");
       end;

julia> for line in eachline("my_file.txt")
           print(line)
       end
JuliaLang is a GitHub organization. It has many members.

julia> rm("my_file.txt");
```

source

Base.displaysize – Function.

```
displaysize([io::IO]) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this `IO` object. If no input is provided, the environment variables `LINES` and `COLUMNS` are read. If those are not set, a default size of `(24, 80)` is returned.

Examples

```
julia> withenv("LINES" => 30, "COLUMNS" => 100) do
           displaysize()
       end
(30, 100)
```

To get your TTY size,

```
julia> displaysize(stdout)
(34, 147)
```

source

57.3 Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function `display(x)` to request the richest available multimedia display of a Julia object `x` (with a plain-text fallback).
- Overloading `show` allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.
- Multimedia-capable display backends may be registered by subclassing a generic `AbstractDisplay` type and pushing them onto a stack of display backends via `pushdisplay`.

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

`Base.Multimedia.AbstractDisplay` — Type.

```
AbstractDisplay
```

Abstract supertype for rich display output devices. `TextDisplay` is a subtype of this.

source

`Base.Multimedia.display` – Function.

```
display(x)
display(d::AbstractDisplay, x)
display(mime, x)
display(d::AbstractDisplay, mime, x)
```

AbstractDisplay `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text `stdout` output as a fallback. The `display(d, x)` variant attempts to display `x` on the given display `d` only, throwing a `MethodError` if `d` cannot display objects of this type.

In general, you cannot assume that `display` output goes to `stdout` (unlike `print(x)` or `show(x)`). For example, `display(x)` may open up a separate window with an image. `display(x)` means "show `x` in the best way you can for the current output device(s)." If you want REPL-like text output that is guaranteed to go to `stdout`, use `show(stdout, "text/plain", x)` instead.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display `x` using the requested MIME type only, throwing a `MethodError` if this type is not supported by either the display(s) or by `x`. With these variants, one can also supply the "raw" data in the requested MIME type by passing `x::AbstractString` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{UInt8}` (for binary MIME types).

To customize how instances of a type are displayed, overload `show` rather than `display`, as explained in the manual section on custom pretty-printing.

source

`Base.Multimedia.redisplay` – Function.

```
redisplay(x)
redisplay(d::AbstractDisplay, x)
redisplay(mime, x)
redisplay(d::AbstractDisplay, mime, x)
```

By default, the `redisplay` functions simply call `display`. However, some display backends may override `redisplay` to modify an existing display of `x` (if any). Using `redisplay` is also a hint to the backend that `x` may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

source

Base.Multimedia.displayable – Function.

```
displayable(mime) -> Bool
displayable(d::AbstractDisplay, mime) -> Bool
```

Returns a boolean value indicating whether the given `mime` type (string) is displayable by any of the displays in the current display stack, or specifically by the display `d` in the second variant.

source

Missing docstring.

Missing docstring for `Base.show(::Any, ::Any, ::Any)`. Check Documenter's build log for details.

Base.Multimedia.showable – Function.

```
showable(mime, x)
```

Returns a boolean value indicating whether or not the object `x` can be written as the given `mime` type.

(By default, this is determined automatically by the existence of the corresponding show method for `typeof(x)`. Some types provide custom `showable` methods; for example, if the available MIME formats depend on the value of `x`.)

Examples

```
julia> showable(MIME("text/plain"), rand(5))
true

julia> showable("img/png", rand(5))
false
```

source

Base.repr – Method.

```
repr(mime, x; context=nothing)
```

Returns an `AbstractString` or `Vector{UInt8}` containing the representation of `x` in the requested `mime` type, as written by show(io, mime, x) (throwing a MethodError if no appropriate `show` is available). An `AbstractString` is returned for MIME types with textual representations (such as `"text/html"` or `"application/postscript"`),

whereas binary data is returned as `Vector{UInt8}`. (The function `istextmime(mime)` returns whether or not Julia treats a given `mime` type as text.)

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

As a special case, if `x` is an `AbstractString` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `repr` function assumes that `x` is already in the requested `mime` format and simply returns `x`. This special case does not apply to the "`text/plain`" MIME type. This is useful so that raw data can be passed to `display(m::MIME, x)`.

In particular, `repr("text/plain", x)` is typically a "pretty-printed" version of `x` designed for human consumption. See also `repr(x)` to instead return a string corresponding to `show(x)` that may be closer to how the value of `x` would be entered in Julia.

Examples

```
julia> A = [1 2; 3 4];

julia> repr("text/plain", A)
"2×2 Array{Int64,2}:\n 1  2\n 3  4"
```

source

Base.Multimedia.MIME – Type.

```
MIME
```

A type representing a standard internet data format. "MIME" stands for "Multipurpose Internet Mail Extensions", since the standard was originally used to describe multimedia attachments to email messages.

A `MIME` object can be passed as the second argument to `show` to request output in that format.

Examples

```
julia> show(stdout, MIME("text/plain"), "hi")
"hi"
```

source

Base.Multimedia.@MIME_str – Macro.

```
@MIME_str
```

A convenience macro for writing MIME types, typically used when adding methods to show. For example the syntax `show(io::IO, ::MIME"text/html", x::MyType) = ...` could be used to define how to write an HTML representation of `MyType`.

source

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling `display(x)` on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype `D` of the abstract class `AbstractDisplay`. Then, for each MIME type (`mime` string) that can be displayed on `D`, one should define a function `display(d::D, ::MIME"mime", x) = ...` that displays `x` as that MIME type, usually by calling `show(io, mime, x)` or `repr(io, mime, x)`. A `MethodError` should be thrown if `x` cannot be displayed as that MIME type; this is automatic if one calls `show` or `repr`. Finally, one should define a function `display(d::D, x)` that queries `showable(mime, x)` for the `mime` types supported by `D` and displays the "best" one; a `MethodError` should be thrown if no supported MIME types are found for `x`. Similarly, some subtypes may wish to override `redisplay(d::D, ...)`. (Again, one should `import Base.display` to add new methods to `display`.) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display "handle" of some type). The display functions for `D` can then be called directly, but they can also be invoked automatically from `display(x)` simply by pushing a new display onto the display-backend stack with:

Base.Multimedia.pushdisplay – Function.

```
pushdisplay(d::AbstractDisplay)
```

Pushes a new display `d` on top of the global display-backend stack. Calling `display(x)` or `display(mime, x)` will display `x` on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a `MethodError`).

source

Base.Multimedia.popdisplay – Function.

```
popdisplay()
popdisplay(d::AbstractDisplay)
```

Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

source

Base.Multimedia.TextDisplay – Type.

```
TextDisplay(io::IO)
```

Returns a `TextDisplay <: AbstractDisplay`, which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

source

`Base.Multimedia.istextmime` – Function.

```
istextmime(m::MIME)
```

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

```
julia> istextmime(MIME("text/plain"))
true

julia> istextmime(MIME("img/png"))
false
```

source

57.4 Network I/O

`Base.bytesavailable` – Function.

```
bytesavailable(io)
```

Return the number of bytes available for reading before a read from this stream or buffer will block.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> bytesavailable(io)
34
```

source

`Base.ntoh` – Function.

```
ntoh(x)
```

Convert the endianness of a value from Network byte order (big-endian) to that used by the Host.

source

Base.hton – Function.

```
hton(x)
```

Convert the endianness of a value from that used by the Host to Network byte order (big-endian).

source

Base.ltoh – Function.

```
ltoh(x)
```

Convert the endianness of a value from Little-endian to that used by the Host.

source

Base.htol – Function.

```
htol(x)
```

Convert the endianness of a value from that used by the Host to Little-endian.

source

Base.ENDIAN_BOM – Constant.

```
ENDIAN_BOM
```

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value `0x04030201`. Big-endian machines will contain the value `0x01020304`.

source

Chapter 58

Punctuation

Extended documentation for mathematical symbols & functions is here.

sym-bol	meaning
`@m`	invoke macro `m`; followed by space-separated expressions
`!`	prefix "not" (logical negation) operator
`a!( )`	at the end of a function name, `!` is used as a convention to indicate that a function modifies its argument(s)
`#`	begin single line comment
`#=`	begin multi-line comment (these are nestable)
`=#`	end multi-line comment
`$`	string and expression interpolation
`%`	remainder operator
`^`	exponent operator
`&`	bitwise and
`&&`	short-circuiting boolean and
`¦`	bitwise or
`¦¦`	short-circuiting boolean or
`⊻`	bitwise xor operator
`*`	multiply, or matrix multiply
`()`	the empty tuple
`~`	bitwise not operator
`\`	backslash operator
`'`	complex transpose operator A⌧
`a[]`	array indexing (calling `getindex` or `setindex!`)
`[,]`	vector literal constructor (calling `vect`)
`[;]`	vertical concatenation (calling `vcat` or `hvcat`)
`[ ]`	with space-separated expressions, horizontal concatenation (calling `hcat` or `hvcat`)
`T{ }`	parametric type instantiation
`;`	statement separator
`,`	separate function arguments or tuple components
`?`	3-argument conditional operator (used like: `conditional ? if_true : if_false`)
`""`	delimit string literals
`''`	delimit character literals
`` ` ` ``	delimit external process (command) specifications
`...`	splice arguments into a function call or declare a varargs function
`.`	access named fields in objects/modules (calling `getproperty` or `setproperty!`), also prefixes elementwise function calls (calling `broadcast`)

Chapter 59

Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. By default, Julia picks reasonable algorithms and sorts in standard ascending order:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

You can easily sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

To sort an array in-place, use the "bang" version of the sort function:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
```

```
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

```
julia> v = randn(5)
5-element Array{Float64,1}:
  0.297288
  0.382396
 -0.597634
 -0.0104452
 -0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Arrays can easily be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

If needed, the sorting algorithm can be chosen:

```
julia> sort(v, alg=InsertionSort)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

All the sorting and order related functions rely on a "less than" relation defining a total order on the values to be manipulated. The `isless` function is invoked by default, but the relation can be specified via the `lt` keyword.

59.1 Sorting Functions

`Base.sort!` — Function.

```
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Sort the vector `v` in place. `QuickSort` is used by default for numeric arrays while `MergeSort` is used for other arrays. You can specify an algorithm to use via the `alg` keyword (see Sorting Algorithms for available algorithms). The `by` keyword lets you provide a function that will be applied to each element before comparison; the `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both `by` and `lt` are specified, the `lt` function is applied to the result of the `by` function; `rev=true` reverses whatever ordering specified via the `by` and `lt` keywords.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Array{Int64,1}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Array{Int64,1}:
 3
 2
 1

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
3-element Array{Tuple{Int64,String},1}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Array{Tuple{Int64,String},1}:
 (3, "a")
 (2, "b")
 (1, "c")
```

source

```
sort!(A; dims::Integer, alg::Algorithm=defalg(A), lt=isless, by=identity, rev::Bool=false, order::Ordering=
    Forward)
```

Sort the multidimensional array `A` along dimension `dims`. See `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

> Julia 1.1
>
> This function requires at least Julia 1.1.

Examples

```
julia> A = [4 3; 1 2]
2×2 Array{Int64,2}:
```

```
 4  3
 1  2

julia> sort!(A, dims = 1); A
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort!(A, dims = 2); A
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.sort – Function.

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Variant of sort! that returns a sorted copy of v leaving v itself unmodified.

Examples

```
julia> v = [3, 1, 2];

julia> sort(v)
3-element Array{Int64,1}:
 1
 2
 3

julia> v
3-element Array{Int64,1}:
 3
 1
 2
```

source

```
sort(A; dims::Integer, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::
     Ordering=Forward)
```

Sort a multidimensional array `A` along the given dimension. See `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

Examples

```
julia> A = [4 3; 1 2]
2×2 Array{Int64,2}:
 4  3
 1  2

julia> sort(A, dims = 1)
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort(A, dims = 2)
2×2 Array{Int64,2}:
 3  4
 1  2
```

source

`Base.sortperm` — Function.

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Return a permutation vector `I` that puts `v[I]` in sorted order. The order is specified using the same keywords as `sort!`. The permutation is guaranteed to be stable even if the sorting algorithm is unstable, meaning that indices of equal elements appear in ascending order.

See also `sortperm!`.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Array{Int64,1}:
 2
 3
```

```
 1


julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

source

Base.Sort.InsertionSort – Constant.

```
InsertionSort
```

Indicate that a sorting function should use the insertion sort algorithm. Insertion sort traverses the collection one element at a time, inserting each element into its correct, sorted position in the output list.

Characteristics:

- stable: preserves the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- in-place in memory.
- quadratic performance in the number of elements to be sorted: it is well-suited to small collections but should not be used for large ones.

source

Base.Sort.MergeSort – Constant.

```
MergeSort
```

Indicate that a sorting function should use the merge sort algorithm. Merge sort divides the collection into subcollections and repeatedly merges them, sorting each subcollection at each step, until the entire collection has been recombined in sorted form.

Characteristics:

- stable: preserves the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- not in-place in memory.

- divide-and-conquer sort strategy.

source

Base.Sort.QuickSort – Constant.

```
QuickSort
```

Indicate that a sorting function should use the quick sort algorithm, which is not stable.

Characteristics:

- not stable: does not preserve the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- in-place in memory.
- divide-and-conquer: sort strategy similar to MergeSort.
- good performance for large collections.

source

Base.Sort.PartialQuickSort – Type.

```
PartialQuickSort{T <: Union{Integer,OrdinalRange}}
```

Indicate that a sorting function should use the partial quick sort algorithm. Partial quick sort returns the smallest k elements sorted from smallest to largest, finding them and sorting them using QuickSort.

Characteristics:

- not stable: does not preserve the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- in-place in memory.
- divide-and-conquer: sort strategy similar to MergeSort.

source

Base.Sort.sortperm! – Function.

```
sortperm!(ix, v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward, initialized::Bool=false)
```

Like sortperm, but accepts a preallocated index vector ix. If initialized is false (the default), ix is initialized to contain the values 1:length(v).

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Array{Int64,1}:
 2
 3
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

source

Base.sortslices – Function.

```
sortslices(A; dims, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort slices of an array A. The required keyword argument dims must be either an integer or a tuple of integers. It specifies the dimension(s) over which the slices are sorted.

E.g., if A is a matrix, dims=1 will sort rows, dims=2 will sort columns. Note that the default comparison function on one dimensional slices sorts lexicographically.

For the remaining keyword arguments, see the documentation of sort!.

Examples

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1) # Sort rows
3×3 Array{Int64,2}:
 -1   6  4
  7   3  5
  9  -2  8
```

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, lt=(x,y)->isless(x[2],y[2]))
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, rev=true)
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2) # Sort columns
3×3 Array{Int64,2}:
  3   5  7
 -1  -4  6
 -2   8  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, alg=InsertionSort, lt=(x,y)->isless(x[2],y[2]))
3×3 Array{Int64,2}:
  5   3  7
 -4  -1  6
  8  -2  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, rev=true)
3×3 Array{Int64,2}:
 7   5   3
 6  -4  -1
 9   8  -2
```

Higher dimensions

`sortslices` extends naturally to higher dimensions. E.g., if `A` is a a 2x2x2 array, `sortslices(A, dims=3)` will sort slices within the 3rd dimension, passing the 2x2 slices `A[:, :, 1]` and `A[:, :, 2]` to the comparison function. Note that while there is no default order on higher-dimensional slices, you may use the `by` or `lt` keyword argument to specify such an order.

If `dims` is a tuple, the order of the dimensions in `dims` is relevant and specifies the linear order of the slices. E.g., if `A` is three dimensional and `dims` is `(1, 2)`, the orderings of the first two dimensions are re-arranged such such that the slices (of the remaining third dimension) are sorted. If `dims` is `(2, 1)` instead, the same slices will be

taken, but the result order will be row-major instead.

Higher dimensional examples

```
julia> A = permutedims(reshape([4 3; 2 1; 'A' 'B'; 'C' 'D'], (2, 2, 2)), (1, 3, 2))
2×2×2 Array{Any,3}:
[:, :, 1] =
 4  3
 2  1

[:, :, 2] =
 'A'  'B'
 'C'  'D'

julia> sortslices(A, dims=(1,2))
2×2×2 Array{Any,3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 'D'  'B'
 'C'  'A'

julia> sortslices(A, dims=(2,1))
2×2×2 Array{Any,3}:
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
 'D'  'C'
 'B'  'A'

julia> sortslices(reshape([5; 4; 3; 2; 1], (1,1,5)), dims=3, by=x->x[1,1])
1×1×5 Array{Int64,3}:
[:, :, 1] =
 1

[:, :, 2] =
 2
```

```
[:, :, 3] =
 3

[:, :, 4] =
 4

[:, :, 5] =
 5
```

source

59.2 Order-Related Functions

Base.issorted – Function.

```
issorted(v, lt=isless, by=identity, rev:Bool=false, order::Ordering=Forward)
```

Test whether a vector is in sorted order. The `lt`, `by` and `rev` keywords modify what order is considered to be sorted just as they do for sort.

Examples

```
julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
false

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true
```

source

Base.Sort.searchsorted – Function.

```
searchsorted(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the range of indices of `a` which compare as equal to `x` (using binary search) according to the order specified by the `by`, `lt` and `rev` keywords, assuming that `a` is already sorted in that order. Return an empty range located at the insertion point if `a` does not contain values equal to `x`.

Examples

```
julia> searchsorted([1, 2, 4, 5, 5, 7], 4) # single match
3:3

julia> searchsorted([1, 2, 4, 5, 5, 7], 5) # multiple matches
4:5

julia> searchsorted([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3:2

julia> searchsorted([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7:6

julia> searchsorted([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1:0
```

source

Base.Sort.searchsortedfirst – Function.

```
searchsortedfirst(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the first value in `a` greater than or equal to `x`, according to the specified order. Return `length(a) + 1` if `x` is greater than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 5) # multiple matches
4

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3
```

```
julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1
```

source

Base.Sort.searchsortedlast – Function.

```
searchsortedlast(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the last value in `a` less than or equal to `x`, according to the specified order. Return `0` if `x` is less than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 5) # multiple matches
5

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
2

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
6

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
0
```

source

Base.Sort.partialsort! – Function.

```
partialsort!(v, k; by=<transform>, lt=<comparison>, rev=false)
```

Partially sort the vector `v` in place, according to the order specified by `by`, `lt` and `rev` so that the value at index `k` (or range of adjacent values if `k` is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If `k` is a single index, that value is returned; if `k` is a range, an array of values at those indices is returned. Note that `partialsort!` does not fully sort the input array.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4)
4

julia> a
5-element Array{Int64,1}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4, rev=true)
2

julia> a
5-element Array{Int64,1}:
 4
 4
 3
 2
 1
```

source

Base.Sort.partialsort – Function.

```
partialsort(v, k, by=<transform>, lt=<comparison>, rev=false)
```

Variant of partialsort! which copies v before partially sorting it, thereby returning the same thing as partialsort! but leaving v unmodified.

source

Base.Sort.partialsortperm – Function.

```
partialsortperm(v, k; by=<transform>, lt=<comparison>, rev=false)
```

Return a partial permutation I of the vector v, so that v[I] returns values of a fully sorted version of v at index k. If k is a range, a vector of indices is returned; if k is an integer, a single index is returned. The order is specified using the same keywords as sort!. The permutation is stable, meaning that indices of equal elements appear in ascending order.

Note that this function is equivalent to, but more efficient than, calling sortperm(...)[k].

Examples

```
julia> v = [3, 1, 2, 1];

julia> v[partialsortperm(v, 1)]
1

julia> p = partialsortperm(v, 1:3)
3-element view(::Array{Int64,1}, 1:3) with eltype Int64:
 2
 4
 3

julia> v[p]
3-element Array{Int64,1}:
 1
 1
 2
```

source

Base.Sort.partialsortperm! – Function.

```
partialsortperm!(ix, v, k; by=<transform>, lt=<comparison>, rev=false, initialized=false)
```

Like partialsortperm, but accepts a preallocated index vector ix the same size as v, which is used to store (a permutation of) the indices of v.

If the index vector ix is initialized with the indices of v (or a permutation thereof), initialized should be set to true.

If initialized is false (the default), then ix is initialized to contain the indices of v.

If initialized is true, but ix does not contain (a permutation of) the indices of v, the behavior of partialsortperm! is undefined.

(Typically, the indices of v will be 1:length(v), although if v has an alternative array type with non-one-based indices, such as an OffsetArray, ix must also be an OffsetArray with the same indices, and must contain as values (a permutation of) these same indices.)

Upon return, ix is guaranteed to have the indices k in their sorted positions, such that

```
partialsortperm!(ix, v, k);
v[ix[k]] == partialsort(v, k)
```

The return value is the kth element of ix if k is an integer, or view into ix if k is a range.

Examples

```
julia> v = [3, 1, 2, 1];

julia> ix = Vector{Int}(undef, 4);

julia> partialsortperm!(ix, v, 1)
2

julia> ix = [1:4;];

julia> partialsortperm!(ix, v, 2:3, initialized=true)
2-element view(::Array{Int64,1}, 2:3) with eltype Int64:
 4
 3
```

source

59.3 Sorting Algorithms

There are currently four sorting algorithms available in base Julia:

- `InsertionSort`
- `QuickSort`
- `PartialQuickSort(k)`
- `MergeSort`

`InsertionSort` is an O(n^2) stable sorting algorithm. It is efficient for very small `n`, and is used internally by `QuickSort`.

`QuickSort` is an O(n log n) sorting algorithm which is in-place, very fast, but not stable – i.e. elements which are considered equal will not remain in the same order in which they originally appeared in the array to be sorted. `QuickSort` is the default algorithm for numeric values, including integers and floats.

`PartialQuickSort(k)` is similar to `QuickSort`, but the output array is only sorted up to index `k` if `k` is an integer, or in the range of `k` if `k` is an `OrdinalRange`. For example:

```
x = rand(1:500, 100)
k = 50
k2 = 50:100
s = sort(x; alg=QuickSort)
ps = sort(x; alg=PartialQuickSort(k))
qs = sort(x; alg=PartialQuickSort(k2))
map(issorted, (s, ps, qs))             # => (true, false, false)
map(x->issorted(x[1:k]), (s, ps, qs))  # => (true, true, false)
map(x->issorted(x[k2]), (s, ps, qs))   # => (true, false, true)
s[1:k] == ps[1:k]                      # => true
s[k2] == qs[k2]                        # => true
```

`MergeSort` is an O(n log n) stable sorting algorithm but is not in-place – it requires a temporary array of half the size of the input array – and is typically not quite as fast as `QuickSort`. It is the default algorithm for non-numeric data.

The default sorting algorithms are chosen on the basis that they are fast and stable, or appear to be so. For numeric types indeed, `QuickSort` is selected as it is faster and indistinguishable in this case from a stable sort (unless the array records its mutations in some way). The stability property comes at a non-negligible cost, so if you don't need it, you may want to explicitly specify your preferred algorithm, e.g. `sort!(v, alg=QuickSort)`.

The mechanism by which Julia picks default sorting algorithms is implemented via the `Base.Sort.defalg` function. It allows a particular algorithm to be registered as the default in all sorting functions for specific arrays. For example, here are the two default methods from `sort.jl`:

```
defalg(v::AbstractArray) = MergeSort
defalg(v::AbstractArray{<:Number}) = QuickSort
```

As for numeric arrays, choosing a non-stable default algorithm for array types for which the notion of a stable sort is meaningless (i.e. when two values comparing equal can not be distinguished) may make sense.

Chapter 60

Iteration utilities

Base.Iterators.Stateful — Type.

```
Stateful(itr)
```

There are several different ways to think about this iterator wrapper:

1. It provides a mutable wrapper around an iterator and its iteration state.
2. It turns an iterator-like abstraction into a `Channel`-like abstraction.
3. It's an iterator that mutates to become its own rest iterator whenever an item is produced.

`Stateful` provides the regular iterator interface. Like other mutable iterators (e.g. `Channel`), if iteration is stopped early (e.g. by a `break` in a `for` loop), iteration can be resumed from the same spot by continuing to iterate over the same iterator object (in contrast, an immutable iterator would restart from the beginning).

Examples

```
julia> a = Iterators.Stateful("abcdef");

julia> isempty(a)
false

julia> popfirst!(a)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(Iterators.take(a, 3))
3-element Array{Char,1}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
```

```
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> collect(a)
2-element Array{Char,1}:
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 'f': ASCII/Unicode U+0066 (category Ll: Letter, lowercase)
```

```
julia> a = Iterators.Stateful([1,1,1,2,3,4]);

julia> for x in a; x == 1 || break; end

julia> peek(a)
3

julia> sum(a) # Sum the remaining elements
7
```

source

Base.Iterators.zip – Function.

```
zip(iters...)
```

Run multiple iterators at the same time, until any of them is exhausted. The value type of the `zip` iterator is a tuple of values of its subiterators.

> Note
>
> `zip` orders the calls to its subiterators in such a way that stateful iterators will not advance when another iterator finishes in the current iteration.

Examples

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Array{String,1}:
 "e"
 "d"
```

```
 "b"
 "c"
 "a"

julia> c = zip(a,b)
zip(1:5, ["e", "d", "b", "c", "a"])

julia> length(c)
5

julia> first(c)
(1, "e")
```

source

Base.Iterators.enumerate – Function.

```
enumerate(iter)
```

An iterator that yields `(i, x)` where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far. Note that `i` may not be valid for indexing `iter`; it's also possible that `x != iter[i]`, if `iter` has indices that do not start at 1. See the `pairs(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
           println("$index $value")
       end
1 a
2 b
3 c
```

source

Base.Iterators.rest – Function.

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

Examples

```
julia> collect(Iterators.rest([1,2,3,4], 2))
3-element Array{Int64,1}:
 2
 3
 4
```

source

Base.Iterators.countfrom – Function.

```
countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

```
julia> for v in Iterators.countfrom(5, 2)
           v > 10 && break
           println(v)
       end
5
7
9
```

source

Base.Iterators.take – Function.

```
take(iter, n)
```

An iterator that generates at most the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
```

```
6-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.take(a,3))
3-element Array{Int64,1}:
 1
 3
 5
```

source

Base.Iterators.drop – Function.

```
drop(iter, n)
```

An iterator that generates all but the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.drop(a,4))
2-element Array{Int64,1}:
  9
 11
```

source

Base.Iterators.cycle – Function.

```
cycle(iter)
```

An iterator that cycles through `iter` forever. If `iter` is empty, so is `cycle(iter)`.

Examples

```
julia> for (i, v) in enumerate(Iterators.cycle("hello"))
           print(v)
           i > 10 && break
       end
hellohelloh
```

source

Base.Iterators.repeated – Function.

```
repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

Examples

```
julia> a = Iterators.repeated([1 2], 4);

julia> collect(a)
4-element Array{Array{Int64,2},1}:
 [1 2]
 [1 2]
 [1 2]
 [1 2]
```

source

Base.Iterators.product – Function.

```
product(iters...)
```

Return an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest.

Examples

```
julia> collect(Iterators.product(1:2, 3:5))
2×3 Array{Tuple{Int64,Int64},2}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)
```

source

Base.Iterators.flatten – Function.

```
flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

```
julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Array{Int64,1}:
 1
 2
 8
 9
```

source

Base.Iterators.partition – Function.

```
partition(collection, n)
```

Iterate over a collection `n` elements at a time.

Examples

```
julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Array{SubArray{Int64,1,Array{Int64,1},Tuple{UnitRange{Int64}},true},1}:
 [1, 2]
 [3, 4]
 [5]
```

source

Base.Iterators.filter – Function.

```
Iterators.filter(flt, itr)
```

Given a predicate function `flt` and an iterable object `itr`, return an iterable object which upon iteration yields the elements `x` of `itr` that satisfy `flt(x)`. The order of the original iterator is preserved.

This function is lazy; that is, it is guaranteed to return in (1) time and use (1) additional space, and `flt` will not be called by an invocation of `filter`. Calls to `flt` will be made when iterating over the returned iterable object. These calls are not cached and repeated calls will be made when reiterating.

See `Base.filter` for an eager implementation of filtering for arrays.

Examples

```
julia> f = Iterators.filter(isodd, [1, 2, 3, 4, 5])
Base.Iterators.Filter{typeof(isodd),Array{Int64,1}}(isodd, [1, 2, 3, 4, 5])

julia> foreach(println, f)
1
3
5
```

source

`Base.Iterators.reverse` – Function.

```
Iterators.reverse(itr)
```

Given an iterator `itr`, then `reverse(itr)` is an iterator over the same collection but in the reverse order.

This iterator is "lazy" in that it does not make a copy of the collection in order to reverse it; see `Base.reverse` for an eager implementation.

Not all iterator types `T` support reverse-order iteration. If `T` doesn't, then iterating over `Iterators.reverse(itr::T)` will throw a `MethodError` because of the missing `iterate` methods for `Iterators.Reverse{T}`. (To implement these methods, the original iterator `itr::T` can be obtained from `r = Iterators.reverse(itr)` by `r.itr`.)

Examples

```
julia> foreach(println, Iterators.reverse(1:5))
5
4
3
2
1
```

```
source
```

Chapter 61

C Interface

ccall – Keyword.

```
ccall((function_name, library), returntype, (argtype1, ...), argvalue1, ...)
ccall(function_name, returntype, (argtype1, ...), argvalue1, ...)
ccall(function_pointer, returntype, (argtype1, ...), argvalue1, ...)
```

Call a function in a C-exported shared library, specified by the tuple `(function_name, library)`, where each component is either a string or symbol. Instead of specifying a library, one can also use a `function_name` symbol or string, which is resolved in the current process. Alternatively, `ccall` may also be used to call a function pointer `function_pointer`, such as one returned by `dlsym`.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `argvalue` to the `ccall` will be converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for unsafe_convert and cconvert for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

source

Core.Intrinsics.cglobal – Function.

```
cglobal((symbol, library) [, type=Cvoid])
```

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in ccall. Returns a `Ptr{Type}`, defaulting to `Ptr{Cvoid}` if no `Type` argument is supplied. The values can be read or written by unsafe_load or unsafe_store!, respectively.

source

Base.@cfunction – Macro.

```
@cfunction(callable, ReturnType, (ArgumentTypes...,)) -> Ptr{Cvoid}
@cfunction($callable, ReturnType, (ArgumentTypes...,)) -> CFunction
```

Generate a C-callable function pointer from the Julia function `callable` for the given type signature. To pass the return value to a `ccall`, use the argument type `Ptr{Cvoid}` in the signature.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression (although it can include a splat expression). And that these arguments will be evaluated in global scope during compile-time (not deferred until runtime). Adding a '$' in front of the function argument changes this to instead create a runtime closure over the local variable `callable` (this is not supported on all architectures).

See manual section on ccall and cfunction usage.

Examples

```
julia> function foo(x::Int, y::Int)
           return x + y
       end

julia> @cfunction(foo, Int, (Int, Int))
Ptr{Cvoid} @0x000000001b82fcd0
```

source

Base.CFunction – Type.

```
CFunction struct
```

Garbage-collection handle for the return value from `@cfunction` when the first argument is annotated with '$'. Like all `cfunction` handles, it should be passed to `ccall` as a `Ptr{Cvoid}`, and will be converted automatically at the call site to the appropriate type.

See @cfunction.

source

Base.unsafe_convert – Function.

```
unsafe_convert(T, x)
```

Convert `x` to a C argument of type `T` where the input `x` must be the return value of `cconvert(T, ...)`.

In cases where `convert` would need to take a Julia object and turn it into a `Ptr`, this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to `x` exists as long as the result of this function will be used. Accordingly, the argument `x` to this function should never be an expression, only a variable name or field reference. For example, `x=a.b.c` is acceptable, but `x=[a,b,c]` is not.

The `unsafe` prefix on this function indicates that using the result of this function after the `x` argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also `cconvert`

source

`Base.cconvert` – Function.

```
cconvert(T,x)
```

Convert `x` to a value to be passed to C code as type `T`, typically by calling `convert(T, x)`.

In cases where `x` cannot be safely converted to `T`, unlike `convert`, `cconvert` may return an object of a type different from `T`, which however is suitable for `unsafe_convert` to handle. The result of this function should be kept valid (for the GC) until the result of `unsafe_convert` is not needed anymore. This can be used to allocate memory that will be accessed by the `ccall`. If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither `convert` nor `cconvert` should take a Julia object and turn it into a `Ptr`.

source

`Base.unsafe_load` – Function.

```
unsafe_load(p::Ptr{T}, i::Integer=1)
```

Load a value of type `T` from the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1]`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

source

Base.unsafe_store! – Function.

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
```

Store a value of type `T` to the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1] = x`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.unsafe_copyto! – Method.

```
unsafe_copyto!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy `N` elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The `unsafe` prefix on this function indicates that no validation is performed on the pointers `dest` and `src` to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.unsafe_copyto! – Method.

```
unsafe_copyto!(dest::Array, do, src::Array, so, N)
```

Copy `N` elements from a source array to a destination, starting at offset `so` in the source and `do` in the destination (1-indexed).

The `unsafe` prefix on this function indicates that no validation is performed to ensure that N is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.copyto! – Function.

```
copyto!(dest::AbstractMatrix, src::UniformScaling)
```

Copies a `UniformScaling` onto a matrix.

> Julia 1.1
>
> In Julia 1.0 this method only supported a square destination matrix. Julia 1.1. added support for a rectangular matrix.

```
copyto!(dest, do, src, so, N)
```

Copy `N` elements from collection `src` starting at offset `so`, to array `dest` starting at offset `do`. Return `dest`.

source

```
copyto!(dest::AbstractArray, src) -> dest
```

Copy all elements from collection `src` to array `dest`, whose length must be greater than or equal to the length `n` of `src`. The first `n` elements of `dest` are overwritten, the other elements are left untouched.

Examples

```
julia> x = [1., 0., 3., 0., 5.];

julia> y = zeros(7);

julia> copyto!(y, x);

julia> y
7-element Array{Float64,1}:
 1.0
 0.0
 3.0
 0.0
 5.0
 0.0
 0.0
```

source

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

source

`Base.pointer` – Function.

```
pointer(array [, index])
```

Get the native address of an array or string, optionally at a given location `index`.

This function is "unsafe". Be careful to ensure that a Julia reference to `array` exists as long as this pointer will be used. The `GC.@preserve` macro should be used to protect the `array` argument from garbage collection within a given block of code.

Calling `Ref(array[, index])` is generally preferable to this function as it guarantees validity.

source

`Base.unsafe_wrap` – Method.

```
unsafe_wrap(Array, pointer::Ptr{T}, dims; own = false)
```

Wrap a Julia `Array` object around the data at the address given by `pointer`, without making a copy. The pointer element type `T` determines the array element type. `dims` is either an integer (for a 1d array) or a tuple of the array dimensions. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

This function is labeled "unsafe" because it will crash if `pointer` is not a valid memory address to data of the requested length.

source

`Base.pointer_from_objref` – Function.

```
pointer_from_objref(x)
```

Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.

This function may not be called on immutable objects, since they do not have stable memory addresses.

See also: `unsafe_pointer_to_objref`.

source

`Base.unsafe_pointer_to_objref` – Function.

```
unsafe_pointer_to_objref(p::Ptr)
```

Convert a `Ptr` to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered "unsafe" and should be used with care.

See also: pointer_from_objref.

source

Base.disable_sigint – Function.

```
disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

This is not needed on worker threads (`Threads.threadid() != 1`) since the `InterruptException` will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

source

Base.reenable_sigint – Function.

```
reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of disable_sigint.

source

Base.systemerror – Function.

```
systemerror(sysfunc[, errno::Cint=Libc.errno()])
systemerror(sysfunc, iftrue::Bool)
```

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is `true`

source

Core.Ptr – Type.

```
Ptr{T}
```

A memory address referring to data of type `T`. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

source

`Core.Ref` – Type.

```
Ref{T}
```

An object that safely references data of type `T`. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

In Julia, `Ref` objects are dereferenced (loaded or stored) with `[]`.

Creation of a `Ref` to a value `x` of type `T` is usually written `Ref(x)`. Additionally, for creating interior pointers to containers (such as Array or Ptr), it can be written `Ref(a, i)` for creating a reference to the `i`-th element of `a`.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references.

There is no invalid (NULL) `Ref` in Julia, but a `C_NULL` instance of `Ptr` can be passed to a `ccall` Ref argument.

Use in broadcasting

`Ref` is sometimes used in broadcasting in order to treat the referenced values as a scalar:

```
julia> isa.(Ref([1,2,3]), [Array, Dict, Int])
3-element BitArray{1}:
 1
 0
 0
```

source

`Base.Cchar` – Type.

```
Cchar
```

Equivalent to the native `char` c-type.

source

`Base.Cuchar` – Type.

```
Cuchar
```

Equivalent to the native `unsigned char` c-type (UInt8).

source

Base.Cshort – Type.

```
Cshort
```

Equivalent to the native `signed short` c-type (Int16).

source

Base.Cstring – Type.

```
Cstring
```

A C-style string composed of the native character type Cchars. `Cstring`s are NUL-terminated. For C-style strings composed of the native wide character type, see Cwstring. For more information about string interopability with C, see the manual.

source

Base.Cushort – Type.

```
Cushort
```

Equivalent to the native `unsigned short` c-type (UInt16).

source

Base.Cint – Type.

```
Cint
```

Equivalent to the native `signed int` c-type (Int32).

source

Base.Cuint – Type.

```
Cuint
```

Equivalent to the native `unsigned int` c-type (`UInt32`).

source

`Base.Clong` – Type.

```
Clong
```

Equivalent to the native `signed long` c-type.

source

`Base.Culong` – Type.

```
Culong
```

Equivalent to the native `unsigned long` c-type.

source

`Base.Clonglong` – Type.

```
Clonglong
```

Equivalent to the native `signed long long` c-type (`Int64`).

source

`Base.Culonglong` – Type.

```
Culonglong
```

Equivalent to the native `unsigned long long` c-type (`UInt64`).

source

`Base.Cintmax_t` – Type.

```
Cintmax_t
```

Equivalent to the native `intmax_t` c-type (`Int64`).

source

`Base.Cuintmax_t` – Type.

```
Cuintmax_t
```

Equivalent to the native `uintmax_t` c-type (`UInt64`).

source

`Base.Csize_t` – Type.

```
Csize_t
```

Equivalent to the native `size_t` c-type (`UInt`).

source

`Base.Cssize_t` – Type.

```
Cssize_t
```

Equivalent to the native `ssize_t` c-type.

source

`Base.Cptrdiff_t` – Type.

```
Cptrdiff_t
```

Equivalent to the native `ptrdiff_t` c-type (`Int`).

source

`Base.Cwchar_t` – Type.

```
Cwchar_t
```

Equivalent to the native `wchar_t` c-type (`Int32`).

source

`Base.Cwstring` – Type.

```
Cwstring
```

A C-style string composed of the native wide character type `Cwchar_t`s. `Cwstring`s are NUL-terminated. For C-style strings composed of the native character type, see `Cstring`. For more information about string interopability with C, see the manual.

source

`Base.Cfloat` — Type.

```
Cfloat
```

Equivalent to the native `float` c-type (`Float32`).

source

`Base.Cdouble` — Type.

```
Cdouble
```

Equivalent to the native `double` c-type (`Float64`).

source

Chapter 62

LLVM Interface

Core.Intrinsics.llvmcall – Function.

```
llvmcall(IR::String, ReturnType, (ArgumentType1, ...), ArgumentValue1, ...)
llvmcall((declarations::String, IR::String), ReturnType, (ArgumentType1, ...), ArgumentValue1, ...)
```

Call LLVM IR string in the first argument. Similar to an LLVM function `define` block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `ArgumentValue` to `llvmcall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (See also the documentation for unsafe_convert and cconvert for further details.) In most cases, this simply results in a call to `convert(ArgumentType, ArgumentValue)`.

See `test/llvmcall.jl` for usage examples.

source

Chapter 63

C Standard Library

`Base.Libc.malloc` – Function.

```
malloc(size::Integer) -> Ptr{Cvoid}
```

Call `malloc` from the C standard library.

source

`Base.Libc.calloc` – Function.

```
calloc(num::Integer, size::Integer) -> Ptr{Cvoid}
```

Call `calloc` from the C standard library.

source

`Base.Libc.realloc` – Function.

```
realloc(addr::Ptr, size::Integer) -> Ptr{Cvoid}
```

Call `realloc` from the C standard library.

See warning in the documentation for `free` regarding only using this on memory originally obtained from `malloc`.

source

`Base.Libc.free` – Function.

```
free(addr::Ptr)
```

Call `free` from the C standard library. Only use this on memory obtained from `malloc`, not on pointers retrieved from other C libraries. `Ptr` objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple `libc` libraries exist on the system.

source

`Base.Libc.errno` – Function.

```
errno([code])
```

Get the value of the C library's `errno`. If an argument is specified, it is used to set the value of `errno`.

The value of `errno` is only valid immediately after a `ccall` to a C library routine that sets it. Specifically, you cannot call `errno` at the next prompt in a REPL, because lots of code is executed between prompts.

source

`Base.Libc.strerror` – Function.

```
strerror(n=errno())
```

Convert a system call error code to a descriptive string

source

`Base.Libc.GetLastError` – Function.

```
GetLastError()
```

Call the Win32 `GetLastError` function [only available on Windows].

source

`Base.Libc.FormatMessage` – Function.

```
FormatMessage(n=GetLastError())
```

Convert a Win32 system call error code to a descriptive string [only available on Windows].

source

`Base.Libc.time` – Method.

```
time(t::TmStruct)
```

Converts a `TmStruct` struct to a number of seconds since the epoch.

source

`Base.Libc.strftime` – Function.

```
strftime([format], time)
```

Convert time, given as a number of seconds since the epoch or a `TmStruct`, to a formatted string using the given format. Supported formats are the same as those in the standard C library.

source

`Base.Libc.strptime` – Function.

```
strptime([format], timestr)
```

Parse a formatted time string into a `TmStruct` giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to `time` to convert it to seconds since the epoch, the `isdst` field should be filled in manually. Setting it to `-1` will tell the C library to use the current system settings to determine the timezone.

source

`Base.Libc.TmStruct` – Type.

```
TmStruct([seconds])
```

Convert a number of seconds since the epoch to broken-down format, with fields `sec`, `min`, `hour`, `mday`, `month`, `year`, `wday`, `yday`, and `isdst`.

source

`Base.Libc.flush_cstdio` – Function.

```
flush_cstdio()
```

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

source

`Base.Libc.systemsleep` – Function.

```
systemsleep(s::Real)
```

Suspends execution for **s** seconds. This function does not yield to Julia's scheduler and therefore blocks the Julia thread that it is running on for the duration of the sleep time.

See also: `sleep`

source

Chapter 64

StackTraces

`Base.StackTraces.StackFrame` – Type.

```
StackFrame
```

Stack information representing execution context, with the following fields:

- `func::Symbol`

  The name of the function containing the execution context.

- `linfo::Union{Core.MethodInstance, CodeInfo, Nothing}`

  The MethodInstance containing the execution context (if it could be found).

- `file::Symbol`

  The path to the file containing the execution context.

- `line::Int`

  The line number in the file containing the execution context.

- `from_c::Bool`

  True if the code is from C.

- `inlined::Bool`

  True if the code is from an inlined frame.

- `pointer::UInt64`

  Representation of the pointer to the execution context as returned by `backtrace`.

source

`Base.StackTraces.StackTrace` – Type.

```
StackTrace
```

An alias for `Vector{StackFrame}` provided for convenience; returned by calls to `stacktrace`.

source

`Base.StackTraces.stacktrace` – Function.

```
stacktrace([trace::Vector{Ptr{Cvoid}},] [c_funcs::Bool=false]) -> StackTrace
```

Returns a stack trace in the form of a vector of `StackFrame`s. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, `stacktrace` first calls `backtrace`.

source

The following methods and types in `Base.StackTraces` are not exported and need to be called e.g. as `StackTraces.lookup(ptr)`.

`Base.StackTraces.lookup` – Function.

```
lookup(pointer::Ptr{Cvoid}) -> Vector{StackFrame}
```

Given a pointer to an execution context (usually generated by a call to `backtrace`), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

source

`Base.StackTraces.remove_frames!` – Function.

```
remove_frames!(stack::StackTrace, name::Symbol)
```

Takes a `StackTrace` (a vector of `StackFrames`) and a function name (a `Symbol`) and removes the `StackFrame` specified by the function name from the `StackTrace` (also removing all frames above the specified function). Primarily used to remove `StackTraces` functions from the `StackTrace` prior to returning it.

source

```
remove_frames!(stack::StackTrace, m::Module)
```

Returns the `StackTrace` with all `StackFrame`s from the provided `Module` removed.

source

Chapter 65

SIMD Support

Type `VecElement{T}` is intended for building libraries of SIMD operations. Practical use of it requires using `llvmcall`. The type is defined as:

```
struct VecElement{T}
    value::T
end
```

It has a special compilation rule: a homogeneous tuple of `VecElement{T}` maps to an LLVM `vector` type when `T` is a primitive bits type and the tuple length is in the set {2-6,8-10,16}.

At `-O3`, the compiler might automatically vectorize operations on such tuples. For example, the following program, when compiled with `julia -O3` generates two SIMD addition instructions (`addps`) on x86 systems:

```
const m128 = NTuple{4,VecElement{Float32}}

function add(a::m128, b::m128)
    (VecElement(a[1].value+b[1].value),
     VecElement(a[2].value+b[2].value),
     VecElement(a[3].value+b[3].value),
     VecElement(a[4].value+b[4].value))
end

triple(c::m128) = add(add(c,c),c)

code_native(triple,(m128,))
```

However, since the automatic vectorization cannot be relied upon, future use will mostly be via libraries that use `llvmcall`.

Part III

표준 라이브러리

Chapter 66

Base64

Base64.Base64 – Module.

```
Base64
```

Functionality for base-64 encoded strings and IO.

Base64.Base64EncodePipe – Type.

```
Base64EncodePipe(ostream)
```

Return a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to `ostream`. Calling `close` on the `Base64EncodePipe` stream is necessary to complete the encoding (but does not close `ostream`).

Examples

```
julia> io = IOBuffer();

julia> iob64_encode = Base64EncodePipe(io);

julia> write(iob64_encode, "Hello!")
6

julia> close(iob64_encode);

julia> str = String(take!(io))
"SGVsbG8h"
```

```
julia> String(base64decode(str))
"Hello!"
```

Base64.base64encode – Function.

```
base64encode(writefunc, args...; context=nothing)
base64encode(args...; context=nothing)
```

Given a write-like function `writefunc`, which takes an I/O stream as its first argument, `base64encode(writefunc, args...)` calls `writefunc` to write `args...` to a base64-encoded string, and returns the string. `base64encode(args...)` is equivalent to `base64encode(write, args...)`: it converts its arguments into bytes using the standard write functions and returns the base64-encoded string.

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or IOContext object whose attributes are used for the I/O stream passed to `writefunc` or `write`.

See also base64decode.

Base64.Base64DecodePipe – Type.

```
Base64DecodePipe(istream)
```

Return a new read-only I/O stream, which decodes base64-encoded data read from `istream`.

Examples

```
julia> io = IOBuffer();

julia> iob64_decode = Base64DecodePipe(io);

julia> write(io, "SGVsbG8h")
8

julia> seekstart(io);

julia> String(read(iob64_decode))
"Hello!"
```

Base64.base64decode – Function.

```
base64decode(string)
```

Decode the base64-encoded **string** and returns a **Vector{UInt8}** of the decoded bytes.

See also base64encode.

Examples

```
julia> b = base64decode("SGVsbG8h")
6-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x21

julia> String(b)
"Hello!"
```

Base64.stringmime – Function.

```
stringmime(mime, x; context=nothing)
```

Returns an **AbstractString** containing the representation of **x** in the requested **mime** type. This is similar to repr(mime, x) except that binary data is base64-encoded as an ASCII string.

The optional keyword argument **context** can be set to **:key=>value** pair or an **IO** or IOContext object whose attributes are used for the I/O stream passed to show.

Chapter 67

CRC32c

CRC32c.crc32c – Function.

```
crc32c(data, crc::UInt32=0x00000000)
```

Compute the CRC-32c checksum of the given `data`, which can be an `Array{UInt8}`, a contiguous subarray thereof, or a `String`. Optionally, you can pass a starting `crc` integer to be mixed in with the checksum. The `crc` parameter can be used to compute a checksum on data divided into chunks: performing `crc32c(data2, crc32c(data1))` is equivalent to the checksum of `[data1; data2]`. (Technically, a little-endian checksum is computed.)

There is also a method `crc32c(io, nb, crc)` to checksum `nb` bytes from a stream `io`, or `crc32c(io, crc)` to checksum all the remaining bytes. Hence you can do `open(crc32c, filename)` to checksum an entire file, or `crc32c(seekstart(buf))` to checksum an `IOBuffer` without calling `take!`.

For a `String`, note that the result is specific to the UTF-8 encoding (a different checksum would be obtained from a different Unicode encoding). To checksum an `a::Array` of some other bitstype, you can do `crc32c(reinterpret(UInt8,a))`, but note that the result may be endian-dependent.

CRC32c.crc32c – Method.

```
crc32c(io::IO, [nb::Integer,] crc::UInt32=0x00000000)
```

Read up to `nb` bytes from `io` and return the CRC-32c checksum, optionally mixed with a starting `crc` integer. If `nb` is not supplied, then `io` will be read until the end of the stream.

Chapter 68

Dates

The `Dates` module provides two types for working with dates: `Date` and `DateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TimeType`. The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the `Date` type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both `Date` and `DateTime` are basically immutable `Int64` wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second [1]. The `DateTime` type is not aware of time zones (naive, in Python parlance), analogous to a LocalDateTime in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both `Date` and `DateTime` are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `0000-12-31` is the day before `0001-01-01`, and year `-0001` (yes, negative one for the year) is 2 BC/BCE, year `-0002` is 3 BC/BCE, etc.

68.1 Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

[1] The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that `Date` and `DateTime` are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00

julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

`Date` or `DateTime` parsing is accomplished by the use of format strings. Format strings work by the notion of defining delimited or fixed-width "slots" that contain a period to parse and passing the text to parse and format string to a `Date` or `DateTime` constructor, of the form `Date("2015-01-01","y-m-d")` or `DateTime("20150101","yyyymmdd")`.

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so `"y-m-d"` lets the parser know that between the first and second slots in a date string like `"2014-07-16"`, it should find the `-` character. The `y`, `m`, and `d` characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `"yyyymmdd"` would correspond to a date string like `"20140716"`. The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition `"yyyymm"` from one period character to the next.

Support for text-form month parsing is also supported through the `u` and `U` characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so `u` corresponds to "Jan", "Feb", "Mar", etc. And `U` corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions `dayname` and `monthname`, custom locales can be loaded by passing in the `locale=>Dict{String,Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the `Date(date_string,format_string)` function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a `Dates.DateFormat`, and pass it instead of a raw format string.

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

You can also use the `dateformat""` string macro. This macro creates the `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5
           Date("2015-01-01", dateformat"y-m-d")
       end
```

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

68.2 Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned in the number of

`Day`, and `DateTime` in the number of `Millisecond`. Similarly, comparing `TimeType` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `Int64` values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 734562

julia> dump(dt2)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 730151

julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)

julia> dt - dt2
4411 days
```

```
julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
2000-02-01T00:00:00

julia> dt - dt2
381110400000 milliseconds
```

68.3 Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

```
julia> Dates.yearmonth(t)
(2014, 1)

julia> Dates.monthday(t)
(1, 31)

julia> Dates.yearmonthday(t)
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 735264

julia> t.instant
Dates.UTInstant{Day}(Day(735264))

julia> Dates.value(t)
735264
```

68.4 Query Functions

Query functions provide calendrical information about a `TimeType`. They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"
```

```
julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the `TimeType`'s year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31

julia> Dates.quarterofyear(t)
1

julia> Dates.dayofquarter(t)
31
```

The `dayname` and `monthname` methods can also take an optional `locale` keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely `dayabbr` and `monthabbr`. First the mapping is loaded into the `LOCALES` variable:

```
julia> french_months = ["janvier", "février", "mars", "avril", "mai", "juin",
                        "juillet", "août", "septembre", "octobre", "novembre", "décembre"];

julia> french_monts_abbrev = ["janv","févr","mars","avril","mai","juin",
                              "juil","août","sept","oct","nov","déc"];

julia> french_days = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"];

julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months, french_monts_abbrev, french_days, [""]);
```

The above mentioned functions can then be used to perform the queries:

```
julia> Dates.dayname(t;locale="french")
"vendredi"

julia> Dates.monthname(t;locale="french")
"janvier"

julia> Dates.monthabbr(t;locale="french")
"janv"
```

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr` will error.

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Array{String,1} at index [5]
Stacktrace:
[...]
```

68.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The `Dates` module approach tries to follow the simple principle of trying to change as little as possible when doing `Period` arithmetic. This approach is also often known as calendrical arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the `Dates` module, it gives the result of February 28th. How does it figure that out? I like to think of the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach

is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month first, where we get 2014-02-29, which adjusts down to 2014-02-28, and then add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' types, not their value or positional order; this means `Year` will always be added first, then `Month`, then `Week`, etc. Hence the following does result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01

julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

Tricky? Perhaps. What is an innocent `Dates` user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Day(1):Date(2014,2,3)
Date(2014, 1, 29):Day(1):Date(2014, 2, 3)

julia> collect(dr)
6-element Array{Date,1}:
 Date(2014, 1, 29)
 Date(2014, 1, 30)
 Date(2014, 1, 31)
 Date(2014, 2, 1)
```

```
 Date(2014, 2, 2)
 Date(2014, 2, 3)

julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
Date(2014, 1, 29):Month(1):Date(2014, 7, 29)

julia> collect(dr)
7-element Array{Date,1}:
 Date(2014, 1, 29)
 Date(2014, 2, 28)
 Date(2014, 3, 29)
 Date(2014, 4, 29)
 Date(2014, 5, 29)
 Date(2014, 6, 29)
 Date(2014, 7, 29)
```

68.6 Adjuster Functions

As convenient as date-period arithmetic is, often the kinds of calculations needed on dates take on a calendrical or temporal nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The `Dates` module provides the adjuster API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single `TimeType` as input and return or adjust to the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input to the Monday of the input's week
2014-07-14

julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last day of the input's month
2014-07-31

julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the last day of the input's quarter
2014-09-30
```

The next two higher-order methods, `tonext`, and `toprev`, generalize working with temporal expressions by taking a

`DateFunction` as first argument, along with a starting `TimeType`. A `DateFunction` is just a function, usually anonymous, that takes a single `TimeType` as input and returns a `Bool`, `true` indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday; # Returns true if the day of the week of x is Tuesday

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience method provided for day of the week adjustments
2014-07-15
```

This is useful with the do-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x
           # Return true on the 4th Thursday of November (Thanksgiving)
           Dates.dayofweek(x) == Dates.Thursday &&
           Dates.dayofweekofmonth(x) == 4 &&
           Dates.month(x) == Dates.November
       end
2014-11-27
```

The `Base.filter` method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to November
# Date range from January 1st, 2014 to January 1st, 2015
julia> dr = Dates.Date(2014):Day(1):Dates.Date(2015);

julia> filter(dr) do x
           Dates.dayofweek(x) == Dates.Tue &&
           Dates.April <= Dates.month(x) <= Dates.Nov &&
           Dates.dayofweekofmonth(x) == 2
       end
8-element Array{Date,1}:
 Date(2014, 4, 8)
 Date(2014, 5, 13)
 Date(2014, 6, 10)
 Date(2014, 7, 8)
 Date(2014, 8, 12)
 Date(2014, 9, 9)
```

```
Date(2014, 10, 14)
Date(2014, 11, 11)
```

Additional examples and tests are available in `stdlib/Dates/test/adjusters.jl`.

68.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period-Real` arithmetic is available. You can extract the underlying integer with `Dates.value`.

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years

julia> y1 + y2
3 years

julia> div(y3,y2)
5

julia> y3 - y2
8 years

julia> y3 % y2
0 years

julia> div(y3,3) # mirrors integer division
3 years

julia> Dates.value(Dates.Millisecond(10))
10
```

68.8 Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`:

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)
2016-08-07T00:00:00
```

Unlike the numeric `round` method, which breaks ties toward the even number by default, the `TimeTyperound` method uses the `RoundNearestTiesUp` rounding mode. (It's difficult to guess what breaking ties to nearest "even" `TimeType` would entail.) Further details on the available `RoundingMode` s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g., `Dates.Second(30)`) divides evenly into the next largest period (in this case, `Dates.Minute(1)`). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a `DateTime` to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that `2016-07-17T12:00:00` was chosen is that it is 17,676,660 hours after `0000-01-01T00:00:00`, and 17,676,660 is divisible by 10.

As Julia `Date` and `DateTime` values are represented according to the ISO 8601 standard, `0000-01-01T00:00:00` was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of `Date` s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest `P(2)`, where `P` is a `Period` type? In some cases (specifically, when `P <: Dates.TimePeriod`) the answer is clear:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00

julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the `Dates` module.

Chapter 69

API reference

69.1 Dates and Time Types

Dates.Period – Type.

```
Period
Year
Month
Week
Day
Hour
Minute
Second
Millisecond
Microsecond
Nanosecond
```

`Period` types represent discrete, human representations of time.

Dates.CompoundPeriod – Type.

```
CompoundPeriod
```

A `CompoundPeriod` is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, "a year and a day" is not a fixed number of days, but can be expressed using a `CompoundPeriod`. In fact, a `CompoundPeriod` is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a `CompoundPeriod` result.

Dates.Instant – Type.

```
Instant
```

`Instant` types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

Dates.UTInstant – Type.

```
UTInstant{T}
```

The `UTInstant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The `T` is a `Period` parameter that indicates the resolution or precision of the instant.

Dates.TimeType – Type.

```
TimeType
```

`TimeType` types wrap `Instant` machine instances to provide human representations of the machine instant. `Time`, `DateTime` and `Date` are subtypes of `TimeType`.

Dates.DateTime – Type.

```
DateTime
```

`DateTime` wraps a `UTInstant{Millisecond}` and interprets it according to the proleptic Gregorian calendar.

Dates.Date – Type.

```
Date
```

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

Dates.Time – Type.

```
Time
```

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

69.2 Dates Functions

Dates.DateTime – Method.

```
DateTime(y, [m, d, h, mi, s, ms]) -> DateTime
```

Construct a **DateTime** type by parts. Arguments must be convertible to Int64.

Dates.DateTime – Method.

```
DateTime(periods::Period...) -> DateTime
```

Construct a **DateTime** type by **Period** type parts. Arguments may be in any order. DateTime parts not provided will default to the value of **Dates.default(period)**.

Dates.DateTime – Method.

```
DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit=10000) -> DateTime
```

Create a **DateTime** through the adjuster API. The starting point will be constructed from the provided **y, m, d...** arguments, and will be adjusted until **f::Function** returns **true**. The step size in adjusting can be provided manually through the **step** keyword. **limit** provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that **f::Function** is never satisfied).

Examples

```
julia> DateTime(dt -> Dates.second(dt) == 40, 2010, 10, 20, 10; step = Dates.Second(1))
2010-10-20T10:00:40

julia> DateTime(dt -> Dates.hour(dt) == 20, 2010, 10, 20, 10; step = Dates.Hour(1), limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

Dates.DateTime – Method.

```
DateTime(dt::Date) -> DateTime
```

Convert a **Date** to a **DateTime**. The hour, minute, second, and millisecond parts of the new **DateTime** are assumed to be zero.

Dates.DateTime – Method.

```
DateTime(dt::AbstractString, format::AbstractString; locale="english") -> DateTime
```

Construct a **DateTime** by parsing the **dt** date time string following the pattern given in the **format** string.

This method creates a **DateFormat** object each time it is called. If you are parsing many date time strings of the same format, consider creating a DateFormat object once and using that as the second argument instead.

Dates.format – Method.

```
format(dt::TimeType, format::AbstractString; locale="english") -> AbstractString
```

Construct a string by using a `TimeType` object and applying the provided `format`. The following character codes can be used to construct the `format` string:

Code	Examples	Comment
y	6	Numeric year with a fixed width
Y	1996	Numeric year with a minimum width
m	1, 12	Numeric month with a minimum width
u	Jan	Month name shortened to 3-chars according to the `locale`
U	January	Full month name according to the `locale` keyword
d	1, 31	Day of the month with a minimum width
H	0, 23	Hour (24-hour clock) with a minimum width
M	0, 59	Minute with a minimum width
S	0, 59	Second with a minimum width
s	000, 500	Millisecond with a minimum width of 3
e	Mon, Tue	Abbreviated days of the week
E	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of `yyyy-mm` specifies that the code `y` should have a width of four while `m` a width of two. Codes that yield numeric digits have an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `format` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use `format`: "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format "yyyy₩ymm₩m".

Dates.DateFormat – Type.

```
DateFormat(format::AbstractString, locale="english") -> DateFormat
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Code	Matches	Comment
`y`	1996, 96	Returns year of 1996, 0096
`Y`	1996, 96	Returns year of 1996, 0096. Equivalent to `y`
`m`	1, 01	Matches 1 or 2-digit months
`u`	Jan	Matches abbreviated months according to the `locale` keyword
`U`	January	Matches full month names according to the `locale` keyword
`d`	1, 01	Matches 1 or 2-digit days
`H`	00	Matches hours (24-hour clock)
`I`	00	For outputting hours with 12-hour clock
`M`	00	Matches minutes
`S`	00	Matches seconds
`s`	.500	Matches milliseconds
`e`	Mon, Tues	Matches abbreviated days of the week
`E`	Monday	Matches full name days of the week
`p`	AM	Matches AM/PM (case-insensitive)
`yyyymmdd`	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y₩ym₩m".

Note that 12:00AM corresponds 00:00 (midnight), and 12:00PM corresponds to 12:00 (noon). When parsing a time with a `p` specifier, any hour (either `H` or `I`) is interpreted as as a 12-hour clock, so the `I` code is mainly useful for output.

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many times or try the `dateformat""` string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. see `@dateformat_str`.

See `DateTime` and `format` for how to use a DateFormat object to parse and write Date strings respectively.

`Dates.@dateformat_str` – Macro.

```
dateformat"Y-m-d H:M:S"
```

Create a DateFormat object. Similar to `DateFormat("Y-m-d H:M:S")` but creates the DateFormat object once during macro expansion.

See DateFormat for details about format specifiers.

Dates.DateTime – Method.

```
DateTime(dt::AbstractString, df::DateFormat) -> DateTime
```

Construct a `DateTime` by parsing the `dt` date time string following the pattern given in the DateFormat object. Similar to `DateTime(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted date time strings with a pre-created `DateFormat` object.

Dates.Date – Method.

```
Date(y, [m, d]) -> Date
```

Construct a `Date` type by parts. Arguments must be convertible to Int64.

Dates.Date – Method.

```
Date(period::Period...) -> Date
```

Construct a `Date` type by `Period` type parts. Arguments may be in any order. `Date` parts not provided will default to the value of `Dates.default(period)`.

Dates.Date – Method.

```
Date(f::Function, y[, m, d]; step=Day(1), limit=10000) -> Date
```

Create a `Date` through the adjuster API. The starting point will be constructed from the provided `y, m, d` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that `f::Function` is never satisfied).

Examples

```
julia> Date(date -> Dates.week(date) == 20, 2010, 01, 01)
2010-05-17

julia> Date(date -> Dates.year(date) == 2010, 2000, 01, 01)
2010-01-01
```

```
julia> Date(date -> Dates.month(date) == 10, 2000, 01, 01; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

Dates.Date – Method.

```
Date(dt::DateTime) -> Date
```

Convert a `DateTime` to a `Date`. The hour, minute, second, and millisecond parts of the `DateTime` are truncated, so only the year, month and day parts are used in construction.

Dates.Date – Method.

```
Date(d::AbstractString, format::AbstractString; locale="english") -> Date
```

Construct a `Date` by parsing the `d` date string following the pattern given in the `format` string.

This method creates a `DateFormat` object each time it is called. If you are parsing many date strings of the same format, consider creating a DateFormat object once and using that as the second argument instead.

Dates.Date – Method.

```
Date(d::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string `d` using a `DateFormat` object `df`.

Dates.Time – Method.

```
Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a `Time` type by parts. Arguments must be convertible to Int64.

Dates.Time – Method.

```
Time(period::TimePeriod...) -> Time
```

Construct a `Time` type by `Period` type parts. Arguments may be in any order. `Time` parts not provided will default to the value of `Dates.default(period)`.

Dates.Time – Method.

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int=10000)
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1), limit::Int=10000)
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h, mi, s, ms, us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

Examples

```
julia> Dates.Time(t -> Dates.minute(t) == 30, 20)
20:30:00

julia> Dates.Time(t -> Dates.minute(t) == 0, 20)
20:00:00

julia> Dates.Time(t -> Dates.hour(t) == 10, 3; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Time` – Method.

```
Time(dt::DateTime) -> Time
```

Convert a `DateTime` to a `Time`. The hour, minute, second, and millisecond parts of the `DateTime` are used to create the new `Time`. Microsecond and nanoseconds are zero by default.

`Dates.now` – Method.

```
now() -> DateTime
```

Return a `DateTime` corresponding to the user's system time including the system timezone locale.

`Dates.now` – Method.

```
now(::Type{UTC}) -> DateTime
```

Return a `DateTime` corresponding to the user's system time as UTC/GMT.

`Base.eps` – Function.

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the machine epsilon of the floating point type `T` (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the relative error of `T`, it is a "dimensionless" quantity like `one`.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

source

```
eps(x::AbstractFloat)
```

Return the unit in last place (ulp) of `x`. This is the distance between consecutive representable floating point values at `x`. In most cases, if the distance on either side of `x` is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \mathsf{eps}(x)/2.$$

Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

source

```
eps(::Type{DateTime}) -> Millisecond
eps(::Type{Date}) -> Day
eps(::Type{Time}) -> Nanosecond
eps(::TimeType) -> Period
```

Return the smallest unit value supported by the `TimeType`.

Examples

```
julia> eps(DateTime)
1 millisecond

julia> eps(Date)
1 day

julia> eps(Time)
1 nanosecond
```

Accessor Functions

Dates.year – Function.

```
year(dt::TimeType) -> Int64
```

The year of a **Date** or **DateTime** as an Int64.

Dates.month – Function.

```
month(dt::TimeType) -> Int64
```

The month of a **Date** or **DateTime** as an Int64.

Dates.week – Function.

```
week(dt::TimeType) -> Int64
```

Return the ISO week date of a **Date** or **DateTime** as an Int64. Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, **week(Date(2005, 1, 1))** is the 53rd week of 2004.

Examples

```
julia> Dates.week(Date(1989, 6, 22))
25

julia> Dates.week(Date(2005, 1, 1))
53

julia> Dates.week(Date(2004, 12, 31))
53
```

Dates.day – Function.

```
day(dt::TimeType) -> Int64
```

The day of month of a **Date** or **DateTime** as an Int64.

Dates.hour – Function.

```
hour(dt::DateTime) -> Int64
```

The hour of day of a **DateTime** as an Int64.

```
hour(t::Time) -> Int64
```

The hour of a **Time** as an Int64.

Dates.minute – Function.

```
minute(dt::DateTime) -> Int64
```

The minute of a **DateTime** as an Int64.

```
minute(t::Time) -> Int64
```

The minute of a **Time** as an Int64.

Dates.second – Function.

```
second(dt::DateTime) -> Int64
```

The second of a **DateTime** as an Int64.

```
second(t::Time) -> Int64
```

The second of a **Time** as an Int64.

Dates.millisecond – Function.

```
millisecond(dt::DateTime) -> Int64
```

The millisecond of a **DateTime** as an Int64.

```
millisecond(t::Time) -> Int64
```

The millisecond of a **Time** as an Int64.

Dates.microsecond – Function.

```
microsecond(t::Time) -> Int64
```

The microsecond of a **Time** as an Int64.

Dates.nanosecond – Function.

```
nanosecond(t::Time) -> Int64
```

The nanosecond of a `Time` as an `Int64`.

`Dates.Year` – Method.

```
Year(v)
```

Construct a `Year` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Month` – Method.

```
Month(v)
```

Construct a `Month` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Week` – Method.

```
Week(v)
```

Construct a `Week` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Day` – Method.

```
Day(v)
```

Construct a `Day` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.Hour` – Method.

```
Hour(dt::DateTime) -> Hour
```

The hour part of a DateTime as a `Hour`.

`Dates.Minute` – Method.

```
Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a `Minute`.

`Dates.Second` – Method.

```
Second(dt::DateTime) -> Second
```

The second part of a DateTime as a `Second`.

Dates.Millisecond – Method.

```
Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a `Millisecond`.

Dates.Microsecond – Method.

```
Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a `Microsecond`.

Dates.Nanosecond – Method.

```
Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a `Nanosecond`.

Dates.yearmonth – Function.

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a `Date` or `DateTime`.

Dates.monthday – Function.

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a `Date` or `DateTime`.

Dates.yearmonthday – Function.

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a `Date` or `DateTime`.

Query Functions

Dates.dayname – Function.

```
dayname(dt::TimeType; locale="english") -> String
dayname(day::Integer; locale="english") -> String
```

Return the full day name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> Dates.dayname(Date("2000-01-01"))
"Saturday"

julia> Dates.dayname(4)
"Thursday"
```

`Dates.dayabbr` – Function.

```
dayabbr(dt::TimeType; locale="english") -> String
dayabbr(day::Integer; locale="english") -> String
```

Return the abbreviated name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> Dates.dayabbr(Date("2000-01-01"))
"Sat"

julia> Dates.dayabbr(3)
"Wed"
```

`Dates.dayofweek` – Function.

```
dayofweek(dt::TimeType) -> Int64
```

Return the day of the week as an `Int64` with `1 = Monday, 2 = Tuesday, etc.`.

Examples

```
julia> Dates.dayofweek(Date("2000-01-01"))
6
```

`Dates.dayofmonth` – Function.

```
dayofmonth(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`Dates.dayofweekofmonth` – Function.

```
dayofweekofmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return which number it is in `dt`'s month. So if the day of the week of `dt` is Monday, then `1 = First Monday of the month, 2 = Second Monday of the month, etc.` In the range 1:5.

Examples

```
julia> Dates.dayofweekofmonth(Date("2000-02-01"))
1

julia> Dates.dayofweekofmonth(Date("2000-02-08"))
2

julia> Dates.dayofweekofmonth(Date("2000-02-15"))
3
```

`Dates.daysofweekinmonth` – Function.

```
daysofweekinmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return the total number of that day of the week in `dt`'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including `dayofweekofmonth(dt) == daysofweekinmonth(dt)` in the adjuster function.

Examples

```
julia> Dates.daysofweekinmonth(Date("2005-01-01"))
5

julia> Dates.daysofweekinmonth(Date("2005-01-04"))
4
```

`Dates.monthname` – Function.

```
monthname(dt::TimeType; locale="english") -> String
monthname(month::Integer, locale="english") -> String
```

Return the full name of the month of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> Dates.monthname(Date("2005-01-04"))
"January"

julia> Dates.monthname(2)
"February"
```

`Dates.monthabbr` – Function.

```
monthabbr(dt::TimeType; locale="english") -> String
monthabbr(month::Integer, locale="english") -> String
```

Return the abbreviated month name of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> Dates.monthabbr(Date("2005-01-04"))
"Jan"

julia> monthabbr(2)
"Feb"
```

`Dates.daysinmonth` – Function.

```
daysinmonth(dt::TimeType) -> Int
```

Return the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

Examples

```
julia> Dates.daysinmonth(Date("2000-01"))
31

julia> Dates.daysinmonth(Date("2001-02"))
28

julia> Dates.daysinmonth(Date("2000-02"))
29
```

Dates.isleapyear – Function.

```
isleapyear(dt::TimeType) -> Bool
```

Return `true` if the year of `dt` is a leap year.

Examples

```
julia> Dates.isleapyear(Date("2004"))
true

julia> Dates.isleapyear(Date("2005"))
false
```

Dates.dayofyear – Function.

```
dayofyear(dt::TimeType) -> Int
```

Return the day of the year for `dt` with January 1st being day 1.

Dates.daysinyear – Function.

```
daysinyear(dt::TimeType) -> Int
```

Return 366 if the year of `dt` is a leap year, otherwise return 365.

Examples

```
julia> Dates.daysinyear(1999)
365

julia> Dates.daysinyear(2000)
366
```

Dates.quarterofyear – Function.

```
quarterofyear(dt::TimeType) -> Int
```

Return the quarter that `dt` resides in. Range of value is 1:4.

Dates.dayofquarter – Function.

```
dayofquarter(dt::TimeType) -> Int
```

Return the day of the current quarter of `dt`. Range of value is 1:92.

Adjuster Functions

Base.trunc – Method.

```
trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

```
julia> trunc(Dates.DateTime("1996-01-01T12:30:00"), Dates.Day)
1996-01-01T00:00:00
```

Dates.firstdayofweek – Function.

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

```
julia> Dates.firstdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-01T00:00:00
```

Dates.lastdayofweek – Function.

```
lastdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Sunday of its week.

Examples

```
julia> Dates.lastdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-07T00:00:00
```

Dates.firstdayofmonth – Function.

```
firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its month.

Examples

```
julia> Dates.firstdayofmonth(DateTime("1996-05-20"))
1996-05-01T00:00:00
```

Dates.lastdayofmonth – Function.

```
lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its month.

Examples

```
julia> Dates.lastdayofmonth(DateTime("1996-05-20"))
1996-05-31T00:00:00
```

Dates.firstdayofyear – Function.

```
firstdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its year.

Examples

```
julia> Dates.firstdayofyear(DateTime("1996-05-20"))
1996-01-01T00:00:00
```

Dates.lastdayofyear – Function.

```
lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its year.

Examples

```
julia> Dates.lastdayofyear(DateTime("1996-05-20"))
1996-12-31T00:00:00
```

Dates.firstdayofquarter – Function.

```
firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its quarter.

Examples

```
julia> Dates.firstdayofquarter(DateTime("1996-05-20"))
1996-04-01T00:00:00

julia> Dates.firstdayofquarter(DateTime("1996-08-20"))
1996-07-01T00:00:00
```

Dates.lastdayofquarter – Function.

```
lastdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its quarter.

Examples

```
julia> Dates.lastdayofquarter(DateTime("1996-05-20"))
1996-06-30T00:00:00

julia> Dates.lastdayofquarter(DateTime("1996-08-20"))
1996-09-30T00:00:00
```

Dates.tonext – Method.

```
tonext(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

Dates.toprev – Method.

```
toprev(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the previous day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the previous `dow`, allowing for no adjustment to occur.

Dates.tofirst – Function.

```
tofirst(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the first `dow` of its month. Alternatively, `of=Year` will adjust to the first `dow` of the year.

Dates.tolast – Function.

```
tolast(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the last `dow` of its month. Alternatively, `of=Year` will adjust to the last `dow` of the year.

Dates.tonext – Method.

```
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

Dates.toprev – Method.

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

Periods

Dates.Period – Method.

```
Year(v)
Month(v)
Week(v)
Day(v)
Hour(v)
Minute(v)
Second(v)
Millisecond(v)
Microsecond(v)
Nanosecond(v)
```

Construct a `Period` type with the given `v` value. Input must be losslessly convertible to an `Int64`.

Dates.CompoundPeriod – Method.

```
CompoundPeriod(periods) -> CompoundPeriod
```

Construct a `CompoundPeriod` from a `Vector` of `Period`s. All `Period`s of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
25 hours

julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
-1 hour, 1 minute

julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
1 month, -2 weeks

julia> Dates.CompoundPeriod(Dates.Minute(50000))
50000 minutes
```

`Dates.value` – Function.

```
Dates.value(x::Period) -> Int64
```

For a given period, return the value associated with that period. For example, `value(Millisecond(10))` returns 10 as an integer.

`Dates.default` – Function.

```
default(p::Period) -> Period
```

Returns a sensible "default" value for the input Period by returning `T(1)` for Year, Month, and Day, and `T(0)` for Hour, Minute, Second, and Millisecond.

Rounding Functions

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`.

`Base.floor` – Method.

```
floor(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` less than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `floor(dt, Dates.Hour)` is a shortcut for `floor(dt, Dates.Hour(1))`.

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01
```

```
julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-06T00:00:00
```

Base.ceil – Method.

```
ceil(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Dates.Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

Base.round – Method.

```
round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Return the `Date` or `DateTime` nearest to `dt` at resolution `p`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, `p` may be a type instead of a value: `round(dt, Dates.Hour)` is a shortcut for `round(dt, Dates.Hour(1))`.

```
julia> round(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

Valid rounding modes for `round(::TimeType, ::Period, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Most `Period` values can also be rounded to a specified resolution:

`Base.floor` – Method.

```
floor(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` down to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `floor(x, Dates.Hour)` is a shortcut for `floor(x, Dates.Hour(1))`.

```
julia> floor(Dates.Day(16), Dates.Week)
2 weeks

julia> floor(Dates.Minute(44), Dates.Minute(15))
30 minutes

julia> floor(Dates.Hour(36), Dates.Day)
1 day
```

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

`Base.ceil` – Method.

```
ceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` up to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `ceil(x, Dates.Hour)` is a shortcut for `ceil(x, Dates.Hour(1))`.

```
julia> ceil(Dates.Day(16), Dates.Week)
3 weeks

julia> ceil(Dates.Minute(44), Dates.Minute(15))
45 minutes
```

```
julia> ceil(Dates.Hour(36), Dates.Day)
2 days
```

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

`Base.round` — Method.

```
round(x::Period, precision::T, [r::RoundingMode]) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 90 minutes to the nearest hour) will be rounded up.

For convenience, `precision` may be a type instead of a value: `round(x, Dates.Hour)` is a shortcut for `round(x, Dates.Hour(1))`.

```
julia> round(Dates.Day(16), Dates.Week)
2 weeks

julia> round(Dates.Minute(44), Dates.Minute(15))
45 minutes

julia> round(Dates.Hour(36), Dates.Day)
2 days
```

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

The following functions are not exported:

`Dates.floorceil` — Function.

```
floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the `floor` and `ceil` of a `Date` or `DateTime` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

```
floorceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> (T, T)
```

Simultaneously return the `floor` and `ceil` of `Period` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

Dates.epochdays2date – Function.

```
epochdays2date(days) -> Date
```

Take the number of days since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `Date`.

Dates.epochms2datetime – Function.

```
epochms2datetime(milliseconds) -> DateTime
```

Take the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `DateTime`.

Dates.date2epochdays – Function.

```
date2epochdays(dt::Date) -> Int64
```

Take the given `Date` and return the number of days since the rounding epoch (`0000-01-01T00:00:00`) as an Int64.

Dates.datetime2epochms – Function.

```
datetime2epochms(dt::DateTime) -> Int64
```

Take the given `DateTime` and return the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) as an Int64.

Conversion Functions

Dates.today – Function.

```
today() -> Date
```

Return the date portion of `now()`.

Dates.unix2datetime – Function.

```
unix2datetime(x) -> DateTime
```

Take the number of seconds since unix epoch `1970-01-01T00:00:00` and convert to the corresponding `DateTime`.

Dates.datetime2unix – Function.

```
datetime2unix(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of seconds since the unix epoch `1970-01-01T00:00:00` as a `Float64`.

`Dates.julian2datetime` – Function.

```
julian2datetime(julian_days) -> DateTime
```

Take the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and return the corresponding `DateTime`.

`Dates.datetime2julian` – Function.

```
datetime2julian(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of Julian calendar days since the julian epoch `-4713-11-24T12:00:00` as a `Float64`.

`Dates.rata2datetime` – Function.

```
rata2datetime(days) -> DateTime
```

Take the number of Rata Die days since epoch `0000-12-31T00:00:00` and return the corresponding `DateTime`.

`Dates.datetime2rata` – Function.

```
datetime2rata(dt::TimeType) -> Int64
```

Return the number of Rata Die days since epoch from the given `Date` or `DateTime`.

Constants

Days of the Week:

Variable	Abbr.	Value (Int)
`Monday`	`Mon`	1
`Tuesday`	`Tue`	2
`Wednesday`	`Wed`	3
`Thursday`	`Thu`	4
`Friday`	`Fri`	5
`Saturday`	`Sat`	6
`Sunday`	`Sun`	7

Months of the Year:

Variable	Abbr.	Value (Int)
`January`	`Jan`	1
`February`	`Feb`	2
`March`	`Mar`	3
`April`	`Apr`	4
`May`	`May`	5
`June`	`Jun`	6
`July`	`Jul`	7
`August`	`Aug`	8
`September`	`Sep`	9
`October`	`Oct`	10
`November`	`Nov`	11
`December`	`Dec`	12

Chapter 70

Delimited Files

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, T::Type, eol::AbstractChar; header=false, skipstart=0, skipblanks=true,
↪  use_mmap, quotes=true, dims, comments=false, comment_char='#')
```

Read a matrix from the source where each line (separated by `eol`) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If `T` is a numeric type, the result is an array of that type, with any non-numeric elements as `NaN` for floating-point types, or zero. Other useful values of `T` include `String`, `AbstractString`, and `Any`.

If `header` is `true`, the first row of data will be read as header and the tuple `(data_cells, header_cells)` is returned instead of only `data_cells`.

Specifying `skipstart` will ignore the corresponding number of initial lines from the input.

If `skipblanks` is `true`, blank lines in the input will be ignored.

If `use_mmap` is `true`, the file specified by `source` is memory mapped for potential speedups. Default is `true` except on Windows. On Windows, you may want to specify `true` if the file is large, and is only read once and not written to.

If `quotes` is `true`, columns enclosed within double-quote (") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If `comments` is `true`, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, eol::AbstractChar; options...)
```

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, T::Type; options...)
```

The end of line delimiter is taken as `\n`.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
```

```
            writedlm(io, [x y], ',')
        end;

julia> readdlm("delim_file.txt", ',', Float64)
4×2 Array{Float64,2}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar; options...)
```

The end of line delimiter is taken as \n. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
            writedlm(io, [x y], ',')
        end;

julia> readdlm("delim_file.txt", ',')
4×2 Array{Float64,2}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> z = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
```

```
            writedlm(io, [x z], ',')
        end;

julia> readdlm("delim_file.txt", ',')
4×2 Array{Any,2}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as \n.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
            writedlm(io, [x y])
        end;

julia> readdlm("delim_file.txt", Int64)
4×2 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8

julia> readdlm("delim_file.txt", Float64)
4×2 Array{Float64,2}:
 1.0  5.0
 2.0  6.0
```

```
 3.0  7.0
 4.0  8.0

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end;

julia> readdlm("delim_file.txt")
4×2 Array{Any,2}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

DelimitedFiles.writedlm – Function.

```
writedlm(f, A, delim='\t'; opts)
```

Write `A` (a vector, matrix, or an iterable collection of iterable rows) as text to `f` (either a filename string or an `IO` stream) using the given delimiter `delim` (which defaults to tab, but can be any printable Julia object, typically a `Char` or `AbstractString`).

For example, two vectors `x` and `y` of the same length can be written as two columns of tab-delimited text to `f` by either `writedlm(f, [x y])` or by `writedlm(f, zip(x, y))`.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

Chapter 71

Distributed Computing

`Distributed.addprocs` – Function.

```
addprocs(manager::ClusterManager; kwargs...) -> List of process identifiers
```

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

To launch workers without blocking the REPL, or the containing function if launching workers programmatically, execute `addprocs` in its own task.

Examples

```
# On busy clusters, call `addprocs` asynchronously
t = @async addprocs(...)
```

```
# Utilize workers as and when they come online
if nprocs() > 1   # Ensure at least one new worker is available
   ....   # perform distributed execution
end
```

```
# Retrieve newly launched worker IDs, or any error messages
if istaskdone(t)   # Check if `addprocs` has completed to ensure `fetch` doesn't block
    if nworkers() == N
        new_pids = fetch(t)
    else
```

```
        fetch(t)
    end
end
```

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> List of process identifiers
```

Add processes on remote machines via SSH. Requires `julia` to be installed in the same location on each node, or to be available via a shared file system.

`machines` is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string `machine_spec` or a tuple - `(machine_spec, count)`.

`machine_spec` is a string of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

`count` is the number of workers to be launched on the specified host. If specified as `:auto` it will launch as many workers as the number of CPU threads on the specific host.

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `multiplex`: if `true` then SSH multiplexing is used for SSH tunneling. Default is `false`.
- `sshflags`: specifies additional ssh options, e.g. `sshflags=`-i /home/foo/bar.pem`
- `max_parallel`: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- `dir`: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)
- `enable_threaded_blas`: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.
- `exename`: name of the `julia` executable. Defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"` as the case may be.
- `exeflags`: additional flags passed to the worker processes.
- `topology`: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
  - `topology=:all_to_all`: All processes are connected to each other. The default.
  - `topology=:master_worker`: Only the driver process, i.e. `pid` 1 connects to the workers. The workers do not connect to each other.

  - `topology=:custom`: The `launch` method of the cluster manager specifies the connection topology via fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster manager identity `ident` will connect to all workers specified in `connect_idents`.
- `lazy`: Applicable only with `topology=:all_to_all`. If `true`, worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is true.

Environment variables :

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable `JULIA_WORKER_TIMEOUT`. The value of `JULIA_WORKER_TIMEOUT` on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

```
addprocs(; kwargs...) -> List of process identifiers
```

Equivalent to `addprocs(Sys.CPU_THREADS; kwargs...)`

Note that workers do not run a `.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

```
addprocs(np::Integer; restrict=true, kwargs...) -> List of process identifiers
```

Launches workers using the in-built `LocalManager` which only launches workers on the local host. This can be used to take advantage of multiple cores. `addprocs(4)` will add 4 processes on the local machine. If `restrict` is `true`, binding is restricted to `127.0.0.1`. Keyword args `dir`, `exename`, `exeflags`, `topology`, `lazy` and `enable_threaded_blas` have the same effect as documented for `addprocs(machines)`.

`Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

Examples

```
julia> nprocs()
3


julia> workers()
5-element Array{Int64,1}:
 2
 3
```

Distributed.nworkers – Function.

```
nworkers()
```

Get the number of available worker processes. This is one less than nprocs(). Equal to nprocs() if nprocs() == 1.

Examples

```
$ julia -p 5

julia> nprocs()
6

julia> nworkers()
5
```

Distributed.procs – Method.

```
procs()
```

Return a list of all process identifiers, including pid 1 (which is not included by workers()).

Examples

```
$ julia -p 5

julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
```

Distributed.procs – Method.

```
procs(pid::Integer)
```

Return a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as pid are returned.

Distributed.workers – Function.

```
workers()
```

Return a list of all worker process identifiers.

Examples

```
$ julia -p 5

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

`Distributed.rmprocs` — Function.

```
rmprocs(pids...; waitfor=typemax(Int))
```

Remove the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down:

- If unspecified, `rmprocs` will wait until all requested `pids` are removed.
- An `ErrorException` is raised if all workers cannot be terminated before the requested `waitfor` seconds.
- With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

Examples

```
$ julia -p 5

julia> t = rmprocs(2, 3, waitfor=0)
Task (runnable) @0x0000000107c718d0

julia> wait(t)

julia> workers()
3-element Array{Int64,1}:
 4
 5
 6
```

Distributed.interrupt – Function.

```
interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

```
interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

Distributed.myid – Function.

```
myid()
```

Get the id of the current process.

Examples

```
julia> myid()
1


julia> remotecall_fetch(() -> myid(), 4)
4
```

Distributed.pmap – Function.

```
pmap(f, [::AbstractWorkerPool], c...; distributed=true, batch_size=1, on_error=nothing, retry_delays=[],
↪  retry_check=nothing) -> collection
```

Transform collection `c` by applying `f` to each element using available workers and tasks.

For multiple collection arguments, apply `f` elementwise.

Note that `f` must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, `pmap` distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using asyncmap. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f,c; ntasks=()->nworkers())`

`pmap` can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops `pmap` from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=identity)
4-element Array{Any,1}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `f` on an element a maximum of 3 times without any delay between retries.

```
pmap(f, c; retry_delays = zeros(3))
```

Example: Retry `f` only if the exception is not of type `InexactError`, with exponentially increasing delays up to 3 times. Return a `NaN` in place for all `InexactError` occurrences.

```
pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN : rethrow()), retry_delays = ExponentialBackOff(n = 3))
```

Distributed.RemoteException — Type.

```
RemoteException(captured)
```

Exceptions on remote computations are captured and rethrown locally. A `RemoteException` wraps the `pid` of the worker and a captured exception. A `CapturedException` captures the remote exception and a serializable form of the call stack when the exception was raised.

Distributed.Future — Type.

```
Future(w::Int, rrid::RRID, v::Union{Some, Nothing}=nothing)
```

A `Future` is a placeholder for a single computation of unknown termination status and time. For multiple potential computations, see `RemoteChannel`. See `remoteref_id` for identifying an `AbstractRemoteRef`.

Distributed.RemoteChannel — Type.

```
RemoteChannel(pid::Integer=myid())
```

Make a reference to a `Channel{Any}(1)` on process `pid`. The default `pid` is the current process.

```
RemoteChannel(f::Function, pid::Integer=myid())
```

Create references to remote channels of a specific size and type. `f` is a function that when executed on `pid` must return an implementation of an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10 on `pid`.

The default `pid` is the current process.

Base.fetch — Method.

```
fetch(x::Future)
```

Wait for and get the value of a `Future`. The fetched value is cached locally. Further calls to `fetch` on the same reference return the cached value. If the remote value is an exception, throws a `RemoteException` which captures the remote exception and backtrace.

Base.fetch — Method.

```
fetch(c::RemoteChannel)
```

Wait for and get a value from a RemoteChannel. Exceptions raised are the same as for a Future. Does not remove the item fetched.

Distributed.remotecall – Method.

```
remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function f asynchronously on the given arguments on the specified process. Return a Future. Keyword arguments, if any, are passed through to f.

Distributed.remotecall_wait – Method.

```
remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster wait(remotecall(...)) in one message on the Worker specified by worker id id. Keyword arguments, if any, are passed through to f.

See also wait and remotecall.

Distributed.remotecall_fetch – Method.

```
remotecall_fetch(f, id::Integer, args...; kwargs...)
```

Perform fetch(remotecall(...)) in one message. Keyword arguments, if any, are passed through to f. Any remote exceptions are captured in a RemoteException and thrown.

See also fetch and remotecall.

Examples

```
$ julia -p 2

julia> remotecall_fetch(sqrt, 2, 4)
2.0

julia> remotecall_fetch(sqrt, 2, -4)
ERROR: On worker 2:
DomainError with -4.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
...
```

Distributed.remote_do – Method.

```
remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes `f` on worker `id` asynchronously. Unlike `remotecall`, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive `remotecall`s to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2)` will serialize the call to `f1`, followed by `f2` and `f3` in that order. However, it is not guaranteed that `f1` is executed before `f3` on worker 2.

Any exceptions thrown by `f` are printed to `stderr` on the remote worker.

Keyword arguments, if any, are passed through to `f`.

`Base.put!` – Method.

```
put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Return the first argument.

`Base.put!` – Method.

```
put!(rr::Future, v)
```

Store a value to a `Future` `rr`. `Future`s are write-once remote references. A `put!` on an already set `Future` throws an `Exception`. All asynchronous remote calls return `Future`s and set the value to the return value of the call upon completion.

`Base.take!` – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a `RemoteChannel` `rr`, removing the value(s) in the process.

`Base.isready` – Method.

```
isready(rr::RemoteChannel, args...)
```

Determine whether a `RemoteChannel` has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a `Future` since they are assigned only once.

Base.isready – Method.

```
isready(rr::Future)
```

Determine whether a Future has a value stored to it.

If the argument Future is owned by a different node, this call will block to wait for the answer. It is recommended to wait for rr in a separate task instead or to use a local Channel as a proxy:

```
p = 1
f = Future(p)
@async put!(f, remotecall_fetch(long_computation, p))
isready(f)  # will not block
```

Distributed.AbstractWorkerPool – Type.

```
AbstractWorkerPool
```

Supertype for worker pools such as WorkerPool and CachingPool. An AbstractWorkerPool should implement:

- push! - add a new worker to the overall pool (available + busy)
- put! - put back a worker to the available pool
- take! - take a worker from the available pool (to be used for remote function execution)
- length - number of workers available in the overall pool
- isready - return false if a take! on the pool would block, else true

The default implementations of the above (on a AbstractWorkerPool) require fields

- channel::Channel{Int}
- workers::Set{Int}

where channel contains free worker pids and workers is the set of all workers associated with this pool.

Distributed.WorkerPool – Type.

```
WorkerPool(workers::Vector{Int})
```

Create a WorkerPool from a vector of worker ids.

Examples

```
$ julia -p 3

julia> WorkerPool([2, 3])
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([2, 3]), RemoteChannel{Channel{Any}}(1, 1,
↪  6))
```

Distributed.CachingPool – Type.

```
CachingPool(workers::Vector{Int})
```

An implementation of an `AbstractWorkerPool`. remote, remotecall_fetch, pmap (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned `CachingPool` object. To clear the cache earlier, use `clear!(pool)`.

For global variables, only the bindings are captured in a closure, not the data. `let` blocks can be used to capture global data.

Examples

```
const foo = rand(10^8);
wp = CachingPool(workers())
let foo = foo
    pmap(wp, i -> sum(foo) + i, 1:100);
end
```

The above would transfer `foo` only once to each worker.

Distributed.default_worker_pool – Function.

```
default_worker_pool()
```

WorkerPool containing idle workers - used by `remote(f)` and pmap (by default).

Examples

```
$ julia -p 3

julia> default_worker_pool()
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:3), Set([4, 2, 3]), RemoteChannel{Channel{Any}}(1,
↪  1, 4))
```

Distributed.clear! – Method.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

Distributed.remote – Function.

```
remote([p::AbstractWorkerPool], f) -> Function
```

Return an anonymous function that executes function `f` on an available worker (drawn from `WorkerPool` `p` if provided) using `remotecall_fetch`.

Distributed.remotecall – Method.

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall(maximum, wp, A)
Future(2, 1, 6, nothing)
```

In this example, the task ran on pid 2, called from pid 1.

Distributed.remotecall_wait – Method.

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall_wait` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall_wait(maximum, wp, A)
Future(3, 1, 9, nothing)

julia> fetch(f)
0.9995177101692958
```

Distributed.remotecall_fetch – Method.

```
remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs...) -> result
```

WorkerPool variant of `remotecall_fetch(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_fetch` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> remotecall_fetch(maximum, wp, A)
0.9995177101692958
```

Distributed.remote_do – Method.

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

WorkerPool variant of `remote_do(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remote_do` on it.

Distributed.@spawnat – Macro.

```
@spawnat p expr
```

Create a closure around an expression and run the closure asynchronously on process `p`. Return a `Future` to the result. If `p` is the quoted literal symbol `:any`, then the system will pick a processor to use automatically.

Examples

```
julia> addprocs(3);

julia> f = @spawnat 2 myid()
Future(2, 1, 3, nothing)

julia> fetch(f)
2

julia> f = @spawnat :any myid()
Future(3, 1, 7, nothing)

julia> fetch(f)
3
```

> Julia 1.3
>
> The `:any` argument is available as of Julia 1.3.

`Distributed.@fetch` – Macro.

```
@fetch expr
```

Equivalent to `fetch(@spawnat :any expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetch myid()
2

julia> @fetch myid()
3

julia> @fetch myid()
4
```

```
julia> @fetch myid()
2
```

Distributed.@fetchfrom – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See fetch and @spawnat.

Examples

```
julia> addprocs(3);

julia> @fetchfrom 2 myid()
2

julia> @fetchfrom 4 myid()
4
```

Distributed.@distributed – Macro.

```
@distributed
```

A distributed memory, parallel for loop of the form :

```
@distributed [reducer] for var = range
    body
end
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, `@distributed` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@distributed` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with @sync, like :

```
@sync @distributed for var = range
    body
end
```

Distributed.@everywhere – Macro.

```
@everywhere [procs()] expr
```

Execute an expression under `Main` on all `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown. For example:

```
@everywhere bar = 1
```

will define `Main.bar` on all current processes. Any processes added later (say with `addprocs()`) will not have the expression defined.

Unlike `@spawnat`, `@everywhere` does not capture any local variables. Instead, local variables can be broadcast using interpolation:

```
foo = 1
@everywhere bar = $foo
```

The optional argument `procs` allows specifying a subset of all processes to have execute the expression.

Equivalent to calling `remotecall_eval(Main, procs, expr)`.

`Distributed.clear!` – Method.

```
clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbol`s . `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

`Distributed.remoteref_id` – Function.

```
remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Future`s and `RemoteChannel`s are identified by fields:

- `where` - refers to the node where the underlying object/storage referred to by the reference actually exists.
- `whence` - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.
- `id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`remoteref_id` is a low-level API which returns a `RRID` object that wraps `whence` and `id` values of a remote reference.

Distributed.channel_from_id – Function.

```
channel_from_id(id) -> c
```

A low-level API which returns the backing `AbstractChannel` for an `id` returned by remoteref_id. The call is valid only on the node where the backing channel exists.

Distributed.worker_id_from_socket – Function.

```
worker_id_from_socket(s) -> pid
```

A low-level API which, given a `IO` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom serialize methods for a type, which optimizes the data written out depending on the receiving process id.

Distributed.cluster_cookie – Method.

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

Distributed.cluster_cookie – Method.

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

71.1 Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in Base: `LocalManager`, for launching additional workers on the same host, and `SSHManager`, for launching on remote hosts via `ssh`. TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

Distributed.ClusterManager – Type.

```
ClusterManager
```

Supertype for cluster managers, which control workers processes as a cluster. Cluster managers implement how workers can be added, removed and communicated with. `SSHManager` and `LocalManager` are subtypes of this.

Distributed.WorkerConfig – Type.

`WorkerConfig`

Type used by ClusterManagers to control workers added to their clusters. Some fields are used by all cluster managers to access a host:

- `io` – the connection used to access the worker (a subtype of `IO` or `Nothing`)
- `host` – the host address (either an `AbstractString` or `Nothing`)
- `port` – the port on the host used to connect to the worker (either an `Int` or `Nothing`)

Some are used by the cluster manager to add workers to an already-initialized host:

- `count` – the number of workers to be launched on the host
- `exename` – the path to the Julia executable on the host, defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"`
- `exeflags` – flags to use when lauching Julia remotely

The `userdata` field is used to store information for each worker by external managers.

Some fields are used by `SSHManager` and similar managers:

- `tunnel` – `true` (use tunneling), `false` (do not use tunneling), or `nothing` (use default for the manager)
- `multiplex` – `true` (use SSH multiplexing for tunneling) or `false`
- `forward` – the forwarding option used for `-L` option of ssh
- `bind_addr` – the address on the remote host to bind to
- `sshflags` – flags to use in establishing the SSH connection
- `max_parallel` – the maximum number of workers to connect to in parallel on the host

Some fields are used by both `LocalManager`s and `SSHManager`s:

- `connect_at` – determines whether this is a worker-to-worker or driver-to-worker setup call
- `process` – the process which will be connected (usually the manager will assign this during `addprocs`)
- `ospid` – the process ID according to the host OS, used to interrupt worker processes
- `environ` – private dictionary used to store temporary information by Local/SSH managers
- `ident` – worker as identified by the `ClusterManager`
- `connect_idents` – list of worker ids the worker must connect to if using a custom topology

- `enable_threaded_blas` – `true`, `false`, or `nothing`, whether to use threaded BLAS or not on the workers

`Distributed.launch` – Function.

```
launch(manager::ClusterManager, params::Dict, launched::Array, launch_ntfy::Condition)
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

`Distributed.manage` – Function.

```
manage(manager::ClusterManager, id::Integer, config::WorkerConfig. op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate `op` values:

- with `:register/:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

`Base.kill` – Method.

```
kill(manager::ClusterManager, pid::Int, config::WorkerConfig)
```

Implemented by cluster managers. It is called on the master process, by `rmprocs`. It should cause the remote worker specified by `pid` to exit. `kill(manager::ClusterManager.....)` executes a remote `exit()` on `pid`.

`Sockets.connect` – Method.

```
connect(manager::ClusterManager, pid::Int, config::WorkerConfig) -> (instrm::IO, outstrm::IO)
```

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id `pid`, specified by `config` and return a pair of `IO` objects. Messages from `pid` to current process will be read off `instrm`, while messages to be sent to `pid` will be written to `outstrm`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `connect(manager::ClusterManager.....)` sets up TCP/IP socket connections in-between workers.

`Distributed.init_worker` – Function.

```
init_worker(cookie::AbstractString, manager::ClusterManager=DefaultClusterManager())
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--worker[=<cookie>]` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `cookie` is a `cluster_cookie`.

`Distributed.start_worker` — Function.

```
start_worker([out::IO=stdout], cookie::AbstractString=readline(stdin); close_stdin::Bool=true,
↪  stderr_to_stdout::Bool=true)
```

`start_worker` is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

host:port information is written to stream `out` (defaults to stdout).

The function reads the cookie from stdin if required, and listens on a free port (or if specified, the port in the `--bind-to` command line option) and schedules tasks to process incoming TCP connections and requests. It also (optionally) closes stdin and redirects stderr to stdout.

It does not return.

`Distributed.process_messages` — Function.

```
process_messages(r_stream::IO, w_stream::IO, incoming::Bool=true)
```

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two `IO` objects, one for incoming messages and the other for messages addressed to the remote worker. If `incoming` is `true`, the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also `cluster_cookie`.

Chapter 72

File Events

`FileWatching.poll_fd` — Function.

```
poll_fd(fd, timeout_s::Real=-1; readable=false, writable=false)
```

Monitor a file descriptor `fd` for changes in the read or write availability, and with a timeout given by `timeout_s` seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to `true`.

The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

`FileWatching.poll_file` — Function.

```
poll_file(path::AbstractString, interval_s::Real=5.007, timeout_s::Real=-1) -> (previous::StatStruct, current)
```

Monitor a file for changes by polling every `interval_s` seconds until a change occurs or `timeout_s` seconds have elapsed. The `interval_s` should be a long period; the default is 5.007 seconds.

Returns a pair of status objects `(previous, current)` when a change is detected. The `previous` status is always a `StatStruct`, but it may have all of the fields zeroed (indicating the file didn't previously exist, or wasn't previously accessible).

The `current` status object may be a `StatStruct`, an `EOFError` (indicating the timeout elapsed), or some other `Exception` subtype (if the `stat` operation failed - for example, if the path does not exist).

To determine when a file was modified, compare `current isa StatStruct && mtime(prev) != mtime(current)` to detect notification of changes. However, using `watch_file` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

FileWatching.watch_file – Function.

```
watch_file(path::AbstractString, timeout_s::Real=-1)
```

Watch file or directory `path` for changes until a change occurs or `timeout_s` seconds have elapsed.

The returned value is an object with boolean fields `changed`, `renamed`, and `timedout`, giving the result of watching the file.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

FileWatching.watch_folder – Function.

```
watch_folder(path::AbstractString, timeout_s::Real=-1)
```

Watches a file or directory `path` for changes until a change has occurred or `timeout_s` seconds have elapsed.

This will continuing tracking changes for `path` in the background until `unwatch_folder` is called on the same `path`.

The returned value is an pair where the first field is the name of the changed file (if available) and the second field is an object with boolean fields `changed`, `renamed`, and `timedout`, giving the event.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

FileWatching.unwatch_folder – Function.

```
unwatch_folder(path::AbstractString)
```

Stop background tracking of changes for `path`. It is not recommended to do this while another task is waiting for `watch_folder` to return on the same path, as the result may be unpredictable.

Chapter 73

Future

The `Future` module implements future behavior of already existing functions, which will replace the current version in a future release of Julia.

`Future.copy!` – Function.

```
Future.copy!(dst, src) -> dst
```

Copy `src` into `dst`.

> Julia 1.1
>
> This function has moved to `Base` with Julia 1.1, consider using `copy!(dst, src)` instead. `Future.copy!` will be deprecated in the future.

`Future.randjump` – Function.

```
randjump(r::MersenneTwister, steps::Integer) -> MersenneTwister
```

Create an initialized `MersenneTwister` object, whose state is moved forward (without generating numbers) from `r` by `steps` steps. One such step corresponds to the generation of two `Float64` numbers. For each different value of `steps`, a large polynomial has to be generated internally. One is already pre-computed for `steps=big(10)^20`.

Chapter 74

Interactive Utilities

Base.Docs.apropos – Function.

```
apropos(string)
```

Search through all documentation for a string, ignoring case.

InteractiveUtils.varinfo – Function.

```
varinfo(m::Module=Main, pattern::Regex=r"")
```

Return a markdown table giving information about exported global variables in a module, optionally restricted to those matching `pattern`.

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

InteractiveUtils.versioninfo – Function.

```
versioninfo(io::IO=stdout; verbose::Bool=false)
```

Print information about the version of Julia in use. The output is controlled with boolean keyword arguments:

- `verbose`: print all additional information

InteractiveUtils.methodswith – Function.

```
methodswith(typ[, module or function]; supertypes::Bool=false])
```

Return an array of methods with an argument of type `typ`.

The optional second argument restricts the search to a particular module or function (the default is all top-level modules).

If keyword `supertypes` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

`InteractiveUtils.subtypes` – Function.

```
subtypes(T::DataType)
```

Return a list of immediate subtypes of DataType `T`. Note that all currently loaded subtypes are included, including those not visible in the current module.

Examples

```
julia> subtypes(Integer)
3-element Array{Any,1}:
 Bool
 Signed
 Unsigned
```

`InteractiveUtils.edit` – Method.

```
edit(path::AbstractString, line::Integer=0)
```

Edit a file or directory optionally providing a line number to edit the file at. Return to the `julia` prompt when you quit the editor. The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

See also: (`define_editor`)[@ref]

`InteractiveUtils.edit` – Method.

```
edit(function, [types])
edit(module)
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. For modules, open the main source file. The module needs to be loaded with `using` or `import` first.

> Julia 1.1
>
> `edit` on modules requires at least Julia 1.1.

To ensure that the file can be opened at the given line, you may need to call `define_editor` first.

`InteractiveUtils.@edit` – Macro.

```
@edit
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `edit` function on the resulting expression.

`InteractiveUtils.less` – Method.

```
less(file::AbstractString, [line::Integer])
```

Show a file using the default pager, optionally providing a starting line number. Returns to the `julia` prompt when you quit the pager.

`InteractiveUtils.less` – Method.

```
less(function, [types])
```

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

`InteractiveUtils.@less` – Macro.

```
@less
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `less` function on the resulting expression.

`InteractiveUtils.@which` – Macro.

```
@which
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the `Method` object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the `which` function.

`InteractiveUtils.@functionloc` – Macro.

```
@functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple `(filename,line)` giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

`InteractiveUtils.@code_lowered` – Macro.

```
@code_lowered
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_lowered` on the resulting expression.

`InteractiveUtils.@code_typed` – Macro.

```
@code_typed
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_typed` on the resulting expression. Use the optional argument `optimize` with

```
@code_typed optimize=true foo(x)
```

to control whether additional optimizations, such as inlining, are also applied.

`InteractiveUtils.code_warntype` – Function.

```
code_warntype([io::IO], f, types; debuginfo=:default)
```

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to `io` which defaults to `stdout`. The ASTs are annotated in such a way as to cause "non-leaf" types to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability. Not all non-leaf types are particularly problematic for performance, so the results need to be used judiciously. In particular, unions containing either `missing` or `nothing` are displayed in yellow, since these are often intentional.

Keyword argument `debuginfo` may be one of `:source` or `:none` (default), to specify the verbosity of code comments.

See `@code_warntype` for more information.

`InteractiveUtils.@code_warntype` – Macro.

```
@code_warntype
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_warntype` on the resulting expression.

InteractiveUtils.code_llvm – Function.

```
code_llvm([io=stdout,], f, types; raw=false, dump_module=false, optimize=true, debuginfo=:default)
```

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to `io`.

If the `optimize` keyword is unset, the code will be shown before LLVM optimizations. All metadata and dbg.* calls are removed from the printed bitcode. For the full IR, set the `raw` keyword to true. To dump the entire module that encapsulates the function (with declarations), set the `dump_module` keyword to true. Keyword argument `debuginfo` may be one of source (default) or none, to specify the verbosity of code comments.

InteractiveUtils.@code_llvm – Macro.

```
@code_llvm
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_llvm on the resulting expression. Set the optional keyword arguments `raw`, `dump_module`, `debuginfo`, `optimize` by putting them and their value before the function call, like this:

```
@code_llvm raw=true dump_module=true debuginfo=:default f(x)
@code_llvm optimize=false f(x)
```

`optimize` controls whether additional optimizations, such as inlining, are also applied. `raw` makes all metadata and dbg.* calls visible. `debuginfo` may be one of `:source` (default) or `:none`, to specify the verbosity of code comments. `dump_module` prints the entire module that encapsulates the function.

InteractiveUtils.code_native – Function.

```
code_native([io=stdout,], f, types; syntax=:att, debuginfo=:default)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to `io`. Switch assembly syntax using `syntax` symbol parameter set to `:att` for AT&T syntax or `:intel` for Intel syntax. Keyword argument `debuginfo` may be one of source (default) or none, to specify the verbosity of code comments.

InteractiveUtils.@code_native – Macro.

```
@code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_native on the resulting expression.

Set the optional keyword argument `debuginfo` by putting it before the function call, like this:

```
@code_native debuginfo=:default f(x)
```

`debuginfo` may be one of `:source` (default) or `:none`, to specify the verbosity of code comments.

`InteractiveUtils.clipboard` – Function.

```
clipboard(x)
```

Send a printed form of `x` to the operating system clipboard ("copy").

```
clipboard() -> AbstractString
```

Return a string with the contents of the operating system clipboard ("paste").

Chapter 75

LibGit2

The LibGit2 module provides bindings to libgit2, a portable C library that implements core functionality for the Git version control system. These bindings are currently used to power Julia's package manager. It is expected that this module will eventually be moved into a separate package.

Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream libgit2 API reference.

`LibGit2.Buffer` — Type.

```
LibGit2.Buffer
```

A data buffer for exporting data from libgit2. Matches the `git_buf` struct.

When fetching data from LibGit2, a typical usage would look like:

```
buf_ref = Ref(Buffer())
@check ccall(..., (Ptr{Buffer},), buf_ref)
# operation on buf_ref
free(buf_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

`LibGit2.CheckoutOptions` — Type.

```
LibGit2.CheckoutOptions
```

Matches the `git_checkout_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `checkout_strategy`: determine how to handle conflicts and whether to force the checkout/recreate missing files.
- `disable_filters`: if nonzero, do not apply filters like CLRF (to convert file newlines between UNIX and DOS).
- `dir_mode`: read/write/access mode for any directories involved in the checkout. Default is `0755`.
- `file_mode`: read/write/access mode for any files involved in the checkout. Default is `0755` or `0644`, depending on the blob.
- `file_open_flags`: bitflags used to open any files during the checkout.
- `notify_flags`: Flags for what sort of conflicts the user should be notified about.
- `notify_cb`: An optional callback function to notify the user if a checkout conflict occurs. If this function returns a non-zero value, the checkout will be cancelled.
- `notify_payload`: Payload for the notify callback function.
- `progress_cb`: An optional callback function to display checkout progress.
- `progress_payload`: Payload for the progress callback.
- `paths`: If not empty, describes which paths to search during the checkout. If empty, the checkout will occur over all files in the repository.
- `baseline`: Expected content of the `workdir`, captured in a (pointer to a) `GitTree`. Defaults to the state of the tree at HEAD.
- `baseline_index`: Expected content of the `workdir`, captured in a (pointer to a) `GitIndex`. Defaults to the state of the index at HEAD.
- `target_directory`: If not empty, checkout to this directory instead of the `workdir`.
- `ancestor_label`: In case of conflicts, the name of the common ancestor side.
- `our_label`: In case of conflicts, the name of "our" side.
- `their_label`: In case of conflicts, the name of "their" side.
- `perfdata_cb`: An optional callback function to display performance data.
- `perfdata_payload`: Payload for the performance callback.

source

`LibGit2.CloneOptions` – Type.

```
LibGit2.CloneOptions
```

Matches the git_clone_options struct.

The fields represent:

- **version**: version of the struct in use, in case this changes later. For now, always **1**.
- **checkout_opts**: The options for performing the checkout of the remote as part of the clone.
- **fetch_opts**: The options for performing the pre-checkout fetch of the remote as part of the clone.
- **bare**: If **0**, clone the full remote repository. If non-zero, perform a bare clone, in which there is no local copy of the source files in the repository and the gitdir and workdir are the same.
- **localclone**: Flag whether to clone a local object database or do a fetch. The default is to let git decide. It will not use the git-aware transport for a local clone, but will use it for URLs which begin with **file://**.
- **checkout_branch**: The name of the branch to checkout. If an empty string, the default branch of the remote will be checked out.
- **repository_cb**: An optional callback which will be used to create the new repository into which the clone is made.
- **repository_cb_payload**: The payload for the repository callback.
- **remote_cb**: An optional callback used to create the GitRemote before making the clone from it.
- **remote_cb_payload**: The payload for the remote callback.

LibGit2.DescribeOptions – Type.

```
LibGit2.DescribeOptions
```

Matches the git_describe_options struct.

The fields represent:

- **version**: version of the struct in use, in case this changes later. For now, always **1**.
- **max_candidates_tags**: consider this many most recent tags in **refs/tags** to describe a commit. Defaults to 10 (so that the 10 most recent tags would be examined to see if they describe a commit).
- **describe_strategy**: whether to consider all entries in **refs/tags** (equivalent to **git-describe --tags**) or all entries in **refs/** (equivalent to **git-describe --all**). The default is to only show annotated tags. If **Consts.DESCRIBE_TAGS** is passed, all tags, annotated or not, will be considered. If **Consts.DESCRIBE_ALL** is passed, any ref in **refs/** will be considered.

- `pattern`: only consider tags which match `pattern`. Supports glob expansion.
- `only_follow_first_parent`: when finding the distance from a matching reference to the described object, only consider the distance from the first parent.
- `show_commit_oid_as_fallback`: if no matching reference can be found which describes a commit, show the commit's `GitHash` instead of throwing an error (the default behavior).

source

`LibGit2.DescribeFormatOptions` – Type.

```
LibGit2.DescribeFormatOptions
```

Matches the `git_describe_format_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `abbreviated_size`: lower bound on the size of the abbreviated `GitHash` to use, defaulting to `7`.
- `always_use_long_format`: set to `1` to use the long format for strings even if a short format can be used.
- `dirty_suffix`: if set, this will be appended to the end of the description string if the `workdir` is dirty.

source

`LibGit2.DiffDelta` – Type.

```
LibGit2.DiffDelta
```

Description of changes to one entry. Matches the `git_diff_delta` struct.

The fields represent:

- `status`: One of `Consts.DELTA_STATUS`, indicating whether the file has been added/modified/deleted.
- `flags`: Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/text, whether they exist on each side of the diff, and whether the object ids are known to be correct.
- `similarity`: Used to indicate if a file has been renamed or copied.
- `nfiles`: The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).
- `old_file`: A `DiffFile` containing information about the file(s) before the changes.

- `new_file`: A `DiffFile` containing information about the file(s) after the changes.

`LibGit2.DiffFile` – Type.

```
LibGit2.DiffFile
```

Description of one side of a delta. Matches the `git_diff_file` struct.

The fields represent:

- `id`: the `GitHash` of the item in the diff. If the item is empty on this side of the diff (for instance, if the diff is of the removal of a file), this will be `GitHash(0)`.
- `path`: a `NULL` terminated path to the item relative to the working directory of the repository.
- `size`: the size of the item in bytes.
- `flags`: a combination of the `git_diff_flag_t` flags. The `i`th bit of this integer sets the `i`th flag.
- `mode`: the `stat` mode for the item.
- `id_abbrev`: only present in LibGit2 versions newer than or equal to `0.25.0`. The length of the `id` field when converted using `string`. Usually equal to `OID_HEXSZ` (40).

`LibGit2.DiffOptionsStruct` – Type.

```
LibGit2.DiffOptionsStruct
```

Matches the `git_diff_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `flags`: flags controlling which files will appear in the diff. Defaults to `DIFF_NORMAL`.
- `ignore_submodules`: whether to look at files in submodules or not. Defaults to `SUBMODULE_IGNORE_UNSPECIFIED`, which means the submodule's configuration will control whether it appears in the diff or not.
- `pathspec`: path to files to include in the diff. Default is to use all files in the repository.
- `notify_cb`: optional callback which will notify the user of changes to the diff as file deltas are added to it.
- `progress_cb`: optional callback which will display diff progress. Only relevant on libgit2 versions at least as new as 0.24.0.
- `payload`: the payload to pass to `notify_cb` and `progress_cb`.

- `context_lines`: the number of unchanged lines used to define the edges of a hunk. This is also the number of lines which will be shown before/after a hunk to provide context. Default is 3.
- `interhunk_lines`: the maximum number of unchanged lines between two separate hunks allowed before the hunks will be combined. Default is 0.
- `id_abbrev`: sets the length of the abbreviated `GitHash` to print. Default is `7`.
- `max_size`: the maximum file size of a blob. Above this size, it will be treated as a binary blob. The default is 512 MB.
- `old_prefix`: the virtual file directory in which to place old files on one side of the diff. Default is `"a"`.
- `new_prefix`: the virtual file directory in which to place new files on one side of the diff. Default is `"b"`.

source

`LibGit2.FetchHead` – Type.

```
LibGit2.FetchHead
```

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

The fields represent:

- `name`: The name in the local reference database of the fetch head, for example, `"refs/heads/master"`.
- `url`: The URL of the fetch head.
- `oid`: The `GitHash` of the tip of the fetch head.
- `ismerge`: Boolean flag indicating whether the changes at the remote have been merged into the local copy yet or not. If `true`, the local copy is up to date with the remote fetch head.

`LibGit2.FetchOptions` – Type.

```
LibGit2.FetchOptions
```

Matches the `git_fetch_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `callbacks`: remote callbacks to use during the fetch.

- `prune`: whether to perform a prune after the fetch or not. The default is to use the setting from the `GitConfig`.
- `update_fetchhead`: whether to update the `FetchHead` after the fetch. The default is to perform the update, which is the normal git behavior.
- `download_tags`: whether to download tags present at the remote or not. The default is to request the tags for objects which are being downloaded anyway from the server.
- `proxy_opts`: options for connecting to the remote through a proxy. See `ProxyOptions`. Only present on libgit2 versions newer than or equal to 0.25.0.
- `custom_headers`: any extra headers needed for the fetch. Only present on libgit2 versions newer than or equal to 0.24.0.

`LibGit2.GitAnnotated` — Type.

```
GitAnnotated(repo::GitRepo, commit_id::GitHash)
GitAnnotated(repo::GitRepo, ref::GitReference)
GitAnnotated(repo::GitRepo, fh::FetchHead)
GitAnnotated(repo::GitRepo, comittish::AbstractString)
```

An annotated git commit carries with it information about how it was looked up and why, so that rebase or merge operations have more information about the context of the commit. Conflict files contain information about the source/target branches in the merge which are conflicting, for instance. An annotated commit can refer to the tip of a remote branch, for instance when a `FetchHead` is passed, or to a branch head described using `GitReference`.

`LibGit2.GitBlame` — Type.

```
GitBlame(repo::GitRepo, path::AbstractString; options::BlameOptions=BlameOptions())
```

Construct a `GitBlame` object for the file at `path`, using change information gleaned from the history of `repo`. The `GitBlame` object records who changed which chunks of the file when, and how. `options` controls how to separate the contents of the file and which commits to probe - see `BlameOptions` for more information.

`LibGit2.GitBlob` — Type.

```
GitBlob(repo::GitRepo, hash::AbstractGitHash)
GitBlob(repo::GitRepo, spec::AbstractString)
```

Return a `GitBlob` object from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitCommit` — Type.

```
GitCommit(repo::GitRepo, hash::AbstractGitHash)
GitCommit(repo::GitRepo, spec::AbstractString)
```

Return a `GitCommit` object from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitHash` — Type.

```
GitHash
```

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a `GitObject` in a repository.

`LibGit2.GitObject` — Type.

```
GitObject(repo::GitRepo, hash::AbstractGitHash)
GitObject(repo::GitRepo, spec::AbstractString)
```

Return the specified object (`GitCommit`, `GitBlob`, `GitTree` or `GitTag`) from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitRemote` — Type.

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString) -> GitRemote
```

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemote(repo, "upstream", repo_url)
```

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString, fetch_spec::AbstractString) ->
     GitRemote
```

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Examples

```
repo = LibGit2.init(repo_path)
refspec = "+refs/heads/mybranch:refs/remotes/origin/mybranch"
remote = LibGit2.GitRemote(repo, "upstream", repo_url, refspec)
```

LibGit2.GitRemoteAnon – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString) -> GitRemote
```

Look up a remote git repository using only its URL, not its name.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemoteAnon(repo, repo_url)
```

LibGit2.GitRepo – Type.

```
LibGit2.GitRepo(path::AbstractString)
```

Open a git repository at `path`.

LibGit2.GitRepoExt – Function.

```
LibGit2.GitRepoExt(path::AbstractString, flags::Cuint = Cuint(Consts.REPOSITORY_OPEN_DEFAULT))
```

Open a git repository at `path` with extended controls (for instance, if the current user must be a member of a special access group to read `path`).

LibGit2.GitRevWalker – Type.

```
GitRevWalker(repo::GitRepo)
```

A `GitRevWalker` walks through the revisions (i.e. commits) of a git repository `repo`. It is a collection of the commits in the repository, and supports iteration and calls to `map` and `count` (for instance, `count` could be used to determine what percentage of commits in a repository were made by a certain author).

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    count((oid,repo)->(oid == commit_oid1), walker, oid=commit_oid1, by=LibGit2.Consts.SORT_TIME)
end
```

Here, `count` finds the number of commits along the walk with a certain `GitHash`. Since the `GitHash` is unique to a commit, `cnt` will be `1`.

LibGit2.GitShortHash – Type.

```
GitShortHash(hash::GitHash, len::Integer)
```

A shortened git object identifier, which can be used to identify a git object when it is unique, consisting of the initial `len` hexadecimal digits of `hash` (the remaining digits are ignored).

LibGit2.GitSignature – Type.

```
LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

LibGit2.GitStatus – Type.

```
LibGit2.GitStatus(repo::GitRepo; status_opts=StatusOptions())
```

Collect information about the status of each file in the git repository `repo` (e.g. is the file modified, staged, etc.). `status_opts` can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not. See `StatusOptions` for more information.

LibGit2.GitTag – Type.

```
GitTag(repo::GitRepo, hash::AbstractGitHash)
GitTag(repo::GitRepo, spec::AbstractString)
```

Return a `GitTag` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

LibGit2.GitTree – Type.

```
GitTree(repo::GitRepo, hash::AbstractGitHash)
GitTree(repo::GitRepo, spec::AbstractString)
```

Return a `GitTree` object from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.IndexEntry` — Type.

```
LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

`LibGit2.IndexTime` — Type.

```
LibGit2.IndexTime
```

Matches the `git_index_time` struct.

`LibGit2.BlameOptions` — Type.

```
LibGit2.BlameOptions
```

Matches the `git_blame_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `flags`: one of `Consts.BLAME_NORMAL` or `Consts.BLAME_FIRST_PARENT` (the other blame flags are not yet implemented by libgit2).
- `min_match_characters`: the minimum number of alphanumeric characters which much change in a commit in order for the change to be associated with that commit. The default is 20. Only takes effect if one of the `Consts.BLAME_*_COPIES` flags are used, which libgit2 does not implement yet.
- `newest_commit`: the `GitHash` of the newest commit from which to look at changes.
- `oldest_commit`: the `GitHash` of the oldest commit from which to look at changes.
- `min_line`: the first line of the file from which to starting blaming. The default is `1`.
- `max_line`: the last line of the file to which to blame. The default is `0`, meaning the last line of the file.

source

`LibGit2.MergeOptions` — Type.

```
LibGit2.MergeOptions
```

Matches the `git_merge_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `flags`: an `enum` for flags describing merge behavior. Defined in `git_merge_flag_t`. The corresponding Julia enum is `GIT_MERGE` and has values:
  - `MERGE_FIND_RENAMES`: detect if a file has been renamed between the common ancestor and the "ours" or "theirs" side of the merge. Allows merges where a file has been renamed.
  - `MERGE_FAIL_ON_CONFLICT`: exit immediately if a conflict is found rather than trying to resolve it.
  - `MERGE_SKIP_REUC`: do not write the REUC extension on the index resulting from the merge.
  - `MERGE_NO_RECURSIVE`: if the commits being merged have multiple merge bases, use the first one, rather than trying to recursively merge the bases.
- `rename_threshold`: how similar two files must to consider one a rename of the other. This is an integer that sets the percentage similarity. The default is 50.
- `target_limit`: the maximum number of files to compare with to look for renames. The default is 200.
- `metric`: optional custom function to use to determine the similarity between two files for rename detection.
- `recursion_limit`: the upper limit on the number of merges of common ancestors to perform to try to build a new virtual merge base for the merge. The default is no limit. This field is only present on libgit2 versions newer than 0.24.0.
- `default_driver`: the merge driver to use if both sides have changed. This field is only present on libgit2 versions newer than 0.25.0.
- `file_favor`: how to handle conflicting file contents for the `text` driver.
  - `MERGE_FILE_FAVOR_NORMAL`: if both sides of the merge have changes to a section, make a note of the conflict in the index which `git checkout` will use to create a merge file, which the user can then reference to resolve the conflicts. This is the default.
  - `MERGE_FILE_FAVOR_OURS`: if both sides of the merge have changes to a section, use the version in the "ours" side of the merge in the index.
  - `MERGE_FILE_FAVOR_THEIRS`: if both sides of the merge have changes to a section, use the version in the "theirs" side of the merge in the index.
  - `MERGE_FILE_FAVOR_UNION`: if both sides of the merge have changes to a section, include each unique line from both sides in the file which is put into the index.
- `file_flags`: guidelines for merging files.

source

LibGit2.ProxyOptions – Type.

```
LibGit2.ProxyOptions
```

Options for connecting through a proxy.

Matches the git_proxy_options struct.

The fields represent:

- version: version of the struct in use, in case this changes later. For now, always 1.
- proxytype: an enum for the type of proxy to use. Defined in git_proxy_t. The corresponding Julia enum is GIT_PROXY and has values:
  - PROXY_NONE: do not attempt the connection through a proxy.
  - PROXY_AUTO: attempt to figure out the proxy configuration from the git configuration.
  - PROXY_SPECIFIED: connect using the URL given in the url field of this struct.

  Default is to auto-detect the proxy type.
- url: the URL of the proxy.
- credential_cb: a pointer to a callback function which will be called if the remote requires authentication to connect.
- certificate_cb: a pointer to a callback function which will be called if certificate verification fails. This lets the user decide whether or not to keep connecting. If the function returns 1, connecting will be allowed. If it returns 0, the connection will not be allowed. A negative value can be used to return errors.
- payload: the payload to be provided to the two callback functions.

Examples

```
julia> fo = LibGit2.FetchOptions(
           proxy_opts = LibGit2.ProxyOptions(url = Cstring("https://my_proxy_url.com")))

julia> fetch(remote, "master", options=fo)
```

source

LibGit2.PushOptions – Type.

```
LibGit2.PushOptions
```

Matches the git_push_options struct.

The fields represent:

- version: version of the struct in use, in case this changes later. For now, always 1.
- parallelism: if a pack file must be created, this variable sets the number of worker threads which will be spawned by the packbuilder. If 0, the packbuilder will auto-set the number of threads to use. The default is 1.
- callbacks: the callbacks (e.g. for authentication with the remote) to use for the push.
- proxy_opts: only relevant if the LibGit2 version is greater than or equal to 0.25.0. Sets options for using a proxy to communicate with a remote. See ProxyOptions for more information.
- custom_headers: only relevant if the LibGit2 version is greater than or equal to 0.24.0. Extra headers needed for the push operation.

LibGit2.RebaseOperation – Type.

```
LibGit2.RebaseOperation
```

Describes a single instruction/operation to be performed during the rebase. Matches the git_rebase_operation struct.

The fields represent:

- optype: the type of rebase operation currently being performed. The options are:
  - REBASE_OPERATION_PICK: cherry-pick the commit in question.
  - REBASE_OPERATION_REWORD: cherry-pick the commit in question, but rewrite its message using the prompt.
  - REBASE_OPERATION_EDIT: cherry-pick the commit in question, but allow the user to edit the commit's contents and its message.
  - REBASE_OPERATION_SQUASH: squash the commit in question into the previous commit. The commit messages of the two commits will be merged.
  - REBASE_OPERATION_FIXUP: squash the commit in question into the previous commit. Only the commit message of the previous commit will be used.
  - REBASE_OPERATION_EXEC: do not cherry-pick a commit. Run a command and continue if the command exits successfully.
- id: the GitHash of the commit being worked on during this rebase step.
- exec: in case REBASE_OPERATION_EXEC is used, the command to run during this step (for instance, running the test suite after each commit).

LibGit2.RebaseOptions – Type.

```
LibGit2.RebaseOptions
```

Matches the `git_rebase_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `quiet`: inform other git clients helping with/working on the rebase that the rebase should be done "quietly". Used for interoperability. The default is `1`.
- `inmemory`: start an in-memory rebase. Callers working on the rebase can go through its steps and commit any changes, but cannot rewind HEAD or update the repository. The `workdir` will not be modified. Only present on libgit2 versions newer than or equal to 0.24.0.
- `rewrite_notes_ref`: name of the reference to notes to use to rewrite the commit notes as the rebase is finished.
- `merge_opts`: merge options controlling how the trees will be merged at each rebase step. Only present on libgit2 versions newer than or equal to 0.24.0.
- `checkout_opts`: checkout options for writing files when initializing the rebase, stepping through it, and aborting it. See `CheckoutOptions` for more information.

source

LibGit2.RemoteCallbacks – Type.

```
LibGit2.RemoteCallbacks
```

Callback settings. Matches the `git_remote_callbacks` struct.

LibGit2.SignatureStruct – Type.

```
LibGit2.SignatureStruct
```

An action signature (e.g. for committers, taggers, etc). Matches the `git_signature` struct.

The fields represent:

- `name`: The full name of the committer or author of the commit.
- `email`: The email at which the committer/author can be contacted.

- `when`: a `TimeStruct` indicating when the commit was authored/committed into the repository.

`LibGit2.StatusEntry` – Type.

```
LibGit2.StatusEntry
```

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the `git_status_entry` struct.

The fields represent:

- `status`: contains the status flags for the file, indicating if it is current, or has been changed in some way in the index or work tree.
- `head_to_index`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in HEAD and in the index.
- `index_to_workdir`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in the index and in the `workdir`.

`LibGit2.StatusOptions` – Type.

```
LibGit2.StatusOptions
```

Options to control how `git_status_foreach_ext()` will issue callbacks. Matches the `git_status_opt_t` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `show`: a flag for which files to examine and in which order. The default is `Consts.STATUS_SHOW_INDEX_AND_WORKDIR`.
- `flags`: flags for controlling any callbacks used in a status call.
- `pathspec`: an array of paths to use for path-matching. The behavior of the path-matching will vary depending on the values of `show` and `flags`.
- The `baseline` is the tree to be used for comparison to the working directory and index; defaults to HEAD.

`source`

`LibGit2.StrArrayStruct` – Type.

```
LibGit2.StrArrayStruct
```

A LibGit2 representation of an array of strings. Matches the git_strarray struct.

When fetching data from LibGit2, a typical usage would look like:

```
sa_ref = Ref(StrArrayStruct())
@check ccall(..., (Ptr{StrArrayStruct},), sa_ref)
res = convert(Vector{String}, sa_ref[])
free(sa_ref)
```

In particular, note that **LibGit2.free** should be called afterward on the **Ref** object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```
strs = String[...]
@check ccall(..., (Ptr{StrArrayStruct},), strs)
```

Note that no call to **free** is required as the data is allocated by Julia.

LibGit2.TimeStruct – Type.

```
LibGit2.TimeStruct
```

Time in a signature. Matches the git_time struct.

LibGit2.add! – Function.

```
add!(repo::GitRepo, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
add!(idx::GitIndex, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
```

Add all the files with paths specified by **files** to the index **idx** (or the index of the **repo**). If the file already exists, the index entry will be updated. If the file does not exist already, it will be newly added into the index. **files** may contain glob patterns which will be expanded and any matching files will be added (unless **INDEX_ADD_DISABLE_PATHSPEC_MATCH** is set, see below). If a file has been ignored (in **.gitignore** or in the config), it will not be added, unless it is already being tracked in the index, in which case it will be updated. The keyword argument **flags** is a set of bit-flags which control the behavior with respect to ignored files:

- **Consts.INDEX_ADD_DEFAULT** - default, described above.
- **Consts.INDEX_ADD_FORCE** - disregard the existing ignore rules and force addition of the file to the index even if it is already ignored.
- **Consts.INDEX_ADD_CHECK_PATHSPEC** - cannot be used at the same time as **INDEX_ADD_FORCE**. Check that each file in **files** which exists on disk is not in the ignore list. If one of the files is ignored, the function will return **EINVALIDSPEC**.

- `Consts.INDEX_ADD_DISABLE_PATHSPEC_MATCH` - turn off glob matching, and only add files to the index which exactly match the paths specified in `files`.

`LibGit2.add_fetch!` – Function.

```
add_fetch!(repo::GitRepo, rmt::GitRemote, fetch_spec::String)
```

Add a fetch refspec for the specified `rmt`. This refspec will contain information about which branch(es) to fetch from.

Examples

```
julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

`LibGit2.add_push!` – Function.

```
add_push!(repo::GitRepo, rmt::GitRemote, push_spec::String)
```

Add a push refspec for the specified `rmt`. This refspec will contain information about which branch(es) to push to.

Examples

```
julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, branch);

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

> Note
>
> You may need to `close` and reopen the `GitRemote` in question after updating its push refspecs in order for the change to take effect and for calls to `push` to work.

`LibGit2.addblob!` – Function.

```
LibGit2.addblob!(repo::GitRepo, path::AbstractString)
```

Read the file at `path` and adds it to the object database of `repo` as a loose blob. Return the `GitHash` of the resulting blob.

Examples

```
hash_str = string(commit_oid)
blob_file = joinpath(repo_path, ".git", "objects", hash_str[1:2], hash_str[3:end])
id = LibGit2.addblob!(repo, blob_file)
```

`LibGit2.author` – Function.

```
author(c::GitCommit)
```

Return the `Signature` of the author of the commit `c`. The author is the person who made changes to the relevant file(s). See also `committer`.

`LibGit2.authors` – Function.

```
authors(repo::GitRepo) -> Vector{Signature}
```

Return all authors of commits to the `repo` repository.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")

println(repo_file, commit_msg)
flush(repo_file)
LibGit2.add!(repo, test_file)
sig = LibGit2.Signature("TEST", "TEST@TEST.COM", round(time(), 0), 0)
commit_oid1 = LibGit2.commit(repo, "commit1"; author=sig, committer=sig)
println(repo_file, randstring(10))
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit2"; author=sig, committer=sig)

# will be a Vector of [sig, sig]
auths = LibGit2.authors(repo)
```

`LibGit2.branch` – Function.

```
branch(repo::GitRepo)
```

Equivalent to `git branch`. Create a new branch from the current HEAD.

`LibGit2.branch!` – Function.

```
branch!(repo::GitRepo, branch_name::AbstractString, commit::AbstractString=""; kwargs...)
```

Checkout a new git branch in the `repo` repository. `commit` is the `GitHash`, in string form, which will be the start of the new branch. If `commit` is an empty string, the current HEAD will be used.

The keyword arguments are:

- `track::AbstractString=""`: the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- `force::Bool=false`: if `true`, branch creation will be forced.
- `set_head::Bool=true`: if `true`, after the branch creation finishes the branch head will be set as the HEAD of `repo`.

Equivalent to `git checkout [-b|-B] <branch_name> [<commit>] [--track <track>]`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.branch!(repo, "new_branch", set_head=false)
```

`LibGit2.checkout!` – Function.

```
checkout!(repo::GitRepo, commit::AbstractString=""; force::Bool=true)
```

Equivalent to `git checkout [-f] --detach <commit>`. Checkout the git commit `commit` (a `GitHash` in string form) in `repo`. If `force` is `true`, force the checkout and discard any current changes. Note that this detaches the current HEAD.

Examples

```
repo = LibGit2.init(repo_path)
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "111
")
end
```

```
LibGit2.add!(repo, "file1")
commit_oid = LibGit2.commit(repo, "add file1")
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "112
")
end
# would fail without the force=true
# since there are modifications to the file
LibGit2.checkout!(repo, string(commit_oid), force=true)
```

`LibGit2.clone` — Function.

```
clone(repo_url::AbstractString, repo_path::AbstractString, clone_opts::CloneOptions)
```

Clone the remote repository at `repo_url` (which can be a remote URL or a path on the local filesystem) to `repo_path` (which must be a path on the local filesystem). Options for the clone, such as whether to perform a bare clone or not, are set by `CloneOptions`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo = LibGit2.clone(repo_url, "/home/me/projects/Example")
```

```
clone(repo_url::AbstractString, repo_path::AbstractString; kwargs...)
```

Clone a remote repository located at `repo_url` to the local filesystem location `repo_path`.

The keyword arguments are:

- `branch::AbstractString=""`: which branch of the remote to clone, if not the default repository branch (usually `master`).
- `isbare::Bool=false`: if `true`, clone the remote as a bare repository, which will make `repo_path` itself the git directory instead of `repo_path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.
- `remote_cb::Ptr{Cvoid}=C_NULL`: a callback which will be used to create the remote before it is cloned. If `C_NULL` (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- `credentials::Creds=nothing`: provides credentials and/or settings when authenticating against a private repository.
- `callbacks::Callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git clone [-b <branch>] [--bare] <repo_url> <repo_path>`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo1 = LibGit2.clone(repo_url, "test_path")
repo2 = LibGit2.clone(repo_url, "test_path", isbare=true)
julia_url = "https://github.com/JuliaLang/julia"
julia_repo = LibGit2.clone(julia_url, "julia_path", branch="release-0.6")
```

`LibGit2.commit` – Function.

```
commit(repo::GitRepo, msg::AbstractString; kwargs...) -> GitHash
```

Wrapper around `git_commit_create`. Create a commit in the repository `repo`. `msg` is the commit message. Return the OID of the new commit.

The keyword arguments are:

- `refname::AbstractString=Consts.HEAD_FILE`: if not NULL, the name of the reference to update to point to the new commit. For example, `"HEAD"` will update the HEAD of the current branch. If the reference does not yet exist, it will be created.
- `author::Signature = Signature(repo)` is a `Signature` containing information about the person who authored the commit.
- `committer::Signature = Signature(repo)` is a `Signature` containing information about the person who committed the commit to the repository. Not necessarily the same as `author`, for instance if `author` emailed a patch to `committer` who committed it.
- `tree_id::GitHash = GitHash()` is a git tree to use to create the commit, showing its ancestry and relationship with any other history. `tree` must belong to `repo`.
- `parent_ids::Vector{GitHash}=GitHash[]` is a list of commits by `GitHash` to use as parent commits for the new one, and may be empty. A commit might have multiple parents if it is a merge commit, for example.

```
LibGit2.commit(rb::GitRebase, sig::GitSignature)
```

Commit the current patch to the rebase `rb`, using `sig` as the committer. Is silent if the commit has already been applied.

`LibGit2.committer` – Function.

```
committer(c::GitCommit)
```

Return the `Signature` of the committer of the commit `c`. The committer is the person who committed the changes originally authored by the author, but need not be the same as the `author`, for example, if the `author` emailed a patch to a `committer` who committed it.

`LibGit2.count` – Function.

```
LibGit2.count(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(), by::Cint=Consts.SORT_NONE,
↪  rev::Bool=false)
```

Using the `GitRevWalker` `walker` to "walk" over every commit in the repository's history, find the number of commits which return `true` when `f` is applied to them. The keyword arguments are: * `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors. * `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPOLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first). * `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    count((oid, repo)->(oid == commit_oid1), walker, oid=commit_oid1, by=LibGit2.Consts.SORT_TIME)
end
```

`count` finds the number of commits along the walk with a certain `GitHash` `commit_oid1`, starting the walk from that commit and moving forwards in time from it. Since the `GitHash` is unique to a commit, `cnt` will be `1`.

`LibGit2.counthunks` – Function.

```
counthunks(blame::GitBlame)
```

Return the number of distinct "hunks" with a file. A hunk may contain multiple lines. A hunk is usually a piece of a file that was added/changed/removed together, for example, a function added to a source file or an inner loop that was optimized out of that function later.

`LibGit2.create_branch` – Function.

```
LibGit2.create_branch(repo::GitRepo, bname::AbstractString, commit_obj::GitCommit; force::Bool=false)
```

Create a new branch in the repository `repo` with name `bname`, which points to commit `commit_obj` (which has to be part of `repo`). If `force` is `true`, overwrite an existing branch named `bname` if it exists. If `force` is `false` and a branch already exists named `bname`, this function will throw an error.

LibGit2.credentials_callback – Function.

```
credential_callback(...) -> Cint
```

A LibGit2 credential callback function which provides different credential acquisition functionality w.r.t. a connection protocol. The `payload_ptr` is required to contain a `LibGit2.CredentialPayload` object which will keep track of state and settings.

The `allowed_types` contains a bitmask of `LibGit2.Consts.GIT_CREDTYPE` values specifying which authentication methods should be attempted.

Credential authentication is done in the following order (if supported):

- SSH agent
- SSH private/public key pair
- Username/password plain text

If a user is presented with a credential prompt they can abort the prompt by typing `^D` (pressing the control key together with the `d` key).

Note: Due to the specifics of the `libgit2` authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, we will keep track of state using the payload.

For addition details see the LibGit2 guide on authenticating against a server.

LibGit2.credentials_cb – Function.

C function pointer for `credentials_callback`

LibGit2.default_signature – Function.

Return signature object. Free it after use.

LibGit2.delete_branch – Function.

```
LibGit2.delete_branch(branch::GitReference)
```

Delete the branch pointed to by `branch`.

LibGit2.diff_files – Function.

```
diff_files(repo::GitRepo, branch1::AbstractString, branch2::AbstractString; kwarg...) -> Vector{AbstractString}
```

Show which files have changed in the git repository `repo` between branches `branch1` and `branch2`.

The keyword argument is:

- `filter::Set{Consts.DELTA_STATUS}=Set([Consts.DELTA_ADDED, Consts.DELTA_MODIFIED, Consts.DELTA_DELETED])`), and it sets options for the diff. The default is to show files added, modified, or deleted.

Return only the names of the files which have changed, not their contents.

Examples

```
LibGit2.branch!(repo, "branch/a")
LibGit2.branch!(repo, "branch/b")
# add a file to repo
open(joinpath(LibGit2.path(repo),"file"),"w") do f
    write(f, "hello repo
")
end
LibGit2.add!(repo, "file")
LibGit2.commit(repo, "add file")
# returns ["file"]
filt = Set([LibGit2.Consts.DELTA_ADDED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
# returns [] because existing files weren't modified
filt = Set([LibGit2.Consts.DELTA_MODIFIED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
```

Equivalent to `git diff --name-only --diff-filter=<filter> <branch1> <branch2>`.

`LibGit2.entryid` – Function.

```
entryid(te::GitTreeEntry)
```

Return the `GitHash` of the object to which `te` refers.

`LibGit2.entrytype` – Function.

```
entrytype(te::GitTreeEntry)
```

Return the type of the object to which `te` refers. The result will be one of the types which `objtype` returns, e.g. a `GitTree` or `GitBlob`.

LibGit2.fetch – Function.

```
fetch(rmt::GitRemote, refspecs; options::FetchOptions=FetchOptions(), msg="")
```

Fetch from the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to fetch. The keyword arguments are:

- `options`: determines the options for the fetch, e.g. whether to prune afterwards. See `FetchOptions` for more information.
- `msg`: a message to insert into the reflogs.

```
fetch(repo::GitRepo; kwargs...)
```

Fetches updates from an upstream of the repository `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: which remote, specified by name, of `repo` to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- `remoteurl::AbstractString=""`: the URL of `remote`. If not specified, will be assumed based on the given name of `remote`.
- `refspecs=AbstractString[]`: determines properties of the fetch.
- `credentials=nothing`: provides credentials and/or settings when authenticating against a private `remote`.
- `callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git fetch [<remoteurl>¦<repo>] [<refspecs>]`.

LibGit2.fetchheads – Function.

```
fetchheads(repo::GitRepo) -> Vector{FetchHead}
```

Return the list of all the fetch heads for `repo`, each represented as a `FetchHead`, including their names, URLs, and merge statuses.

Examples

```
julia> fetch_heads = LibGit2.fetchheads(repo);


julia> fetch_heads[1].name
"refs/heads/master"
```

```
julia> fetch_heads[1].ismerge
true

julia> fetch_heads[2].name
"refs/heads/test_branch"

julia> fetch_heads[2].ismerge
false
```

LibGit2.fetch_refspecs – Function.

```
fetch_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the fetch refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to fetch from.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

LibGit2.fetchhead_foreach_cb – Function.

C function pointer for `fetchhead_foreach_callback`

LibGit2.merge_base – Function.

```
merge_base(repo::GitRepo, one::AbstractString, two::AbstractString) -> GitHash
```

Find a merge base (a common ancestor) between the commits `one` and `two`. `one` and `two` may both be in string form. Return the `GitHash` of the merge base.

LibGit2.merge! – Method.

```
merge!(repo::GitRepo; kwargs...) -> Bool
```

Perform a git merge on the repository `repo`, merging commits with diverging history into the current branch. Return `true` if the merge succeeded, `false` if not.

The keyword arguments are:

- `committish::AbstractString=""`: Merge the named commit(s) in `committish`.
- `branch::AbstractString=""`: Merge the branch `branch` and all its commits since it diverged from the current branch.
- `fastforward::Bool=false`: If `fastforward` is `true`, only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return `false`. This is equivalent to the git CLI option `--ff-only`.
- `merge_opts::MergeOptions=MergeOptions()`: `merge_opts` specifies options for the merge, such as merge strategy in case of conflicts.
- `checkout_opts::CheckoutOptions=CheckoutOptions()`: `checkout_opts` specifies options for the checkout step.

Equivalent to `git merge [--ff-only] [<committish> | <branch>]`.

> Note
>
> If you specify a `branch`, this must be done in reference format, since the string will be turned into a `GitReference`. For example, if you wanted to merge branch `branch_a`, you would call `merge!(repo, branch="refs/heads/branch_a")`.

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}; kwargs...) -> Bool
```

Merge changes from the annotated commits (captured as `GitAnnotated` objects) `anns` into the HEAD of the repository `repo`. The keyword arguments are:

- `merge_opts::MergeOptions = MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See `MergeOptions` for more information.
- `checkout_opts::CheckoutOptions = CheckoutOptions()`: options for how to perform the checkout. See `CheckoutOptions` for more information.

`anns` may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in
LibGit2.merge!(repo, [upst_ann])
```

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}, fastforward::Bool; kwargs...) -> Bool
```

Merge changes from the annotated commits (captured as `GitAnnotated` objects) `anns` into the HEAD of the repository `repo`. If `fastforward` is `true`, only a fastforward merge is allowed. In this case, if conflicts occur, the merge will fail. Otherwise, if `fastforward` is `false`, the merge may produce a conflict file which the user will need to resolve.

The keyword arguments are:

- `merge_opts::MergeOptions = MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See `MergeOptions` for more information.
- `checkout_opts::CheckoutOptions = CheckoutOptions()`: options for how to perform the checkout. See `CheckoutOptions` for more information.

`anns` may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann_1 = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in, fastforward
LibGit2.merge!(repo, [upst_ann_1], true)

# merge conflicts!
upst_ann_2 = LibGit2.GitAnnotated(repo, "branch/b")
# merge the branch in, try to fastforward
LibGit2.merge!(repo, [upst_ann_2], true) # will return false
LibGit2.merge!(repo, [upst_ann_2], false) # will return true
```

`LibGit2.ffmerge!` – Function.

```
ffmerge!(repo::GitRepo, ann::GitAnnotated)
```

Fastforward merge changes into current HEAD. This is only possible if the commit referred to by `ann` is descended from the current HEAD (e.g. if pulling changes from a remote branch which is simply ahead of the local branch tip).

`LibGit2.fullname` – Function.

```
LibGit2.fullname(ref::GitReference)
```

Return the name of the reference pointed to by the symbolic reference `ref`. If `ref` is not a symbolic reference, return an empty string.

`LibGit2.features` – Function.

```
features()
```

Return a list of git features the current version of libgit2 supports, such as threading or using HTTPS or SSH.

`LibGit2.filename` – Function.

```
filename(te::GitTreeEntry)
```

Return the filename of the object on disk to which `te` refers.

`LibGit2.filemode` – Function.

```
filemode(te::GitTreeEntry) -> Cint
```

Return the UNIX filemode of the object on disk to which `te` refers as an integer.

`LibGit2.gitdir` – Function.

```
LibGit2.gitdir(repo::GitRepo)
```

Return the location of the "git" files of `repo`:

- for normal repositories, this is the location of the `.git` folder.
- for bare repositories, this is the location of the repository itself.

See also `workdir`, `path`.

`LibGit2.git_url` – Function.

```
LibGit2.git_url(; kwargs...) -> String
```

Create a string based upon the URL components provided. When the `scheme` keyword is not provided the URL produced will use the alternative scp-like syntax.

Keywords

- `scheme::AbstractString=""`: the URL scheme which identifies the protocol to be used. For HTTP use "http", SSH use "ssh", etc. When `scheme` is not provided the output format will be "ssh" but using the scp-like syntax.
- `username::AbstractString=""`: the username to use in the output if provided.
- `password::AbstractString=""`: the password to use in the output if provided.
- `host::AbstractString=""`: the hostname to use in the output. A hostname is required to be specified.
- `port::Union{AbstractString,Integer}=""`: the port number to use in the output if provided. Cannot be specified when using the scp-like syntax.
- `path::AbstractString=""`: the path to use in the output if provided.

  Warning

  Avoid using passwords in URLs. Unlike the credential objects, Julia is not able to securely zero or destroy the sensitive data after use and the password may remain in memory; possibly to be exposed by an uninitialized memory.

Examples

```
julia> LibGit2.git_url(username="git", host="github.com", path="JuliaLang/julia.git")
"git@github.com:JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="https", host="github.com", path="/JuliaLang/julia.git")
"https://github.com/JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="ssh", username="git", host="github.com", port=2222, path="JuliaLang/julia.git")
"ssh://git@github.com:2222/JuliaLang/julia.git"
```

`LibGit2.@githash_str` – Macro.

```
@githash_str -> AbstractGitHash
```

Construct a git hash object from the given string, returning a `GitShortHash` if the string is shorter than 40 hexadecimal digits, otherwise a `GitHash`.

Examples

```
julia> LibGit2.githash"d114feb74ce633"
GitShortHash("d114feb74ce633")

julia> LibGit2.githash"d114feb74ce63307afe878a5228ad014e0289a85"
GitHash("d114feb74ce63307afe878a5228ad014e0289a85")
```

LibGit2.head – Function.

```
LibGit2.head(repo::GitRepo) -> GitReference
```

Return a `GitReference` to the current HEAD of `repo`.

```
head(pkg::AbstractString) -> String
```

Return current HEAD GitHash of the `pkg` repo as a string.

LibGit2.head! – Function.

```
LibGit2.head!(repo::GitRepo, ref::GitReference) -> GitReference
```

Set the HEAD of `repo` to the object pointed to by `ref`.

LibGit2.head_oid – Function.

```
LibGit2.head_oid(repo::GitRepo) -> GitHash
```

Lookup the object id of the current HEAD of git repository `repo`.

LibGit2.headname – Function.

```
LibGit2.headname(repo::GitRepo)
```

Lookup the name of the current HEAD of git repository `repo`. If `repo` is currently detached, return the name of the HEAD it's detached from.

LibGit2.init – Function.

```
LibGit2.init(path::AbstractString, bare::Bool=false) -> GitRepo
```

Open a new git repository at `path`. If `bare` is `false`, the working tree will be created in `path/.git`. If `bare` is `true`, no working directory will be created.

LibGit2.is_ancestor_of – Function.

```
is_ancestor_of(a::AbstractString, b::AbstractString, repo::GitRepo) -> Bool
```

Return `true` if `a`, a `GitHash` in string form, is an ancestor of `b`, a `GitHash` in string form.

Examples

```
julia> repo = LibGit2.GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file1);

julia> commit_oid1 = LibGit2.commit(repo, "commit1");

julia> LibGit2.add!(repo, test_file2);

julia> commit_oid2 = LibGit2.commit(repo, "commit2");

julia> LibGit2.is_ancestor_of(string(commit_oid1), string(commit_oid2), repo)
true
```

`LibGit2.isbinary` — Function.

```
isbinary(blob::GitBlob) -> Bool
```

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

`LibGit2.iscommit` — Function.

```
iscommit(id::AbstractString, repo::GitRepo) -> Bool
```

Check if commit `id` (which is a `GitHash` in string form) is in the repository.

Examples

```
julia> repo = LibGit2.GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file);

julia> commit_oid = LibGit2.commit(repo, "add test_file");

julia> LibGit2.iscommit(string(commit_oid), repo)
true
```

LibGit2.isdiff – Function.

```
LibGit2.isdiff(repo::GitRepo, treeish::AbstractString, pathspecs::AbstractString=""; cached::Bool=false)
```

Checks if there are any differences between the tree specified by `treeish` and the tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdiff(repo, "HEAD") # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdiff(repo, "HEAD") # now true
```

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

LibGit2.isdirty – Function.

```
LibGit2.isdirty(repo::GitRepo, pathspecs::AbstractString=""; cached::Bool=false) -> Bool
```

Check if there have been any changes to tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdirty(repo) # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdirty(repo) # now true
LibGit2.isdirty(repo, new_file) # now true
```

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

LibGit2.isorphan – Function.

```
LibGit2.isorphan(repo::GitRepo)
```

Check if the current branch is an "orphan" branch, i.e. has no commits. The first commit to this branch will have no parents.

LibGit2.isset – Function.

```
isset(val::Integer, flag::Integer)
```

Test whether the bits of `val` indexed by `flag` are set (`1`) or unset (`0`).

LibGit2.iszero – Function.

```
iszero(id::GitHash) -> Bool
```

Determine whether all hexadecimal digits of the given GitHash are zero.

LibGit2.lookup_branch – Function.

```
lookup_branch(repo::GitRepo, branch_name::AbstractString, remote::Bool=false) -> Union{GitReference, Nothing}
```

Determine if the branch specified by `branch_name` exists in the repository `repo`. If `remote` is `true`, `repo` is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

Return either a `GitReference` to the requested branch if it exists, or nothing if not.

LibGit2.map – Function.

```
LibGit2.map(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(), range::AbstractString="",
↪  by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the GitRevWalker `walker` to "walk" over every commit in the repository's history, apply `f` to each commit in the walk. The keyword arguments are: * `oid`: The GitHash of the commit to begin the walk from. The default is to use push_head! and therefore the HEAD commit and all its ancestors. * `range`: A range of `GitHash`s in the format `oid1..oid2`. `f` will be applied to all commits between the two. * `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPOLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first). * `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
oids = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.map((oid, repo)->string(oid), walker, by=LibGit2.Consts.SORT_TIME)
end
```

Here, `map` visits each commit using the `GitRevWalker` and finds its `GitHash`.

LibGit2.mirror_callback – Function.

Mirror callback function

Function sets `+refs/*:refs/*` refspecs and `mirror` flag for remote reference.

`LibGit2.mirror_cb` – Function.

C function pointer for `mirror_callback`

`LibGit2.message` – Function.

```
message(c::GitCommit, raw::Bool=false)
```

Return the commit message describing the changes made in commit `c`. If `raw` is `false`, return a slightly "cleaned up" message (which has any leading newlines removed). If `raw` is `true`, the message is not stripped of any such newlines.

`LibGit2.merge_analysis` – Function.

```
merge_analysis(repo::GitRepo, anns::Vector{GitAnnotated}) -> analysis, preference
```

Run analysis on the branches pointed to by the annotated branch tips `anns` and determine under what circumstances they can be merged. For instance, if `anns[1]` is simply an ancestor of `ann[2]`, then `merge_analysis` will report that a fast-forward merge is possible.

Return two outputs, `analysis` and `preference`. `analysis` has several possible values: * `MERGE_ANALYSIS_NONE`: it is not possible to merge the elements of `anns`. * `MERGE_ANALYSIS_NORMAL`: a regular merge, when HEAD and the commits that the user wishes to merge have all diverged from a common ancestor. In this case the changes have to be resolved and conflicts may occur. * `MERGE_ANALYSIS_UP_TO_DATE`: all the input commits the user wishes to merge can be reached from HEAD, so no merge needs to be performed. * `MERGE_ANALYSIS_FASTFORWARD`: the input commit is a descendant of HEAD and so no merge needs to be performed - instead, the user can simply checkout the input commit(s). * `MERGE_ANALYSIS_UNBORN`: the HEAD of the repository refers to a commit which does not exist. It is not possible to merge, but it may be possible to checkout the input commits. `preference` also has several possible values: * `MERGE_PREFERENCE_NONE`: the user has no preference. * `MERGE_PREFERENCE_NO_FASTFORWARD`: do not allow any fast-forward merges. * `MERGE_PREFERENCE_FASTFORWARD_ONLY`: allow only fast-forward merges and no other type (which may introduce conflicts). `preference` can be controlled through the repository or global git configuration.

`LibGit2.name` – Function.

```
LibGit2.name(ref::GitReference)
```

Return the full name of `ref`.

```
name(rmt::GitRemote)
```

Get the name of a remote repository, for instance `"origin"`. If the remote is anonymous (see `GitRemoteAnon`) the name will be an empty string `""`.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.clone(cache_repo, "test_directory");

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> name(remote)
"origin"
```

```
LibGit2.name(tag::GitTag)
```

The name of `tag` (e.g. `"v0.5"`).

`LibGit2.need_update` – Function.

```
need_update(repo::GitRepo)
```

Equivalent to `git update-index`. Return `true` if `repo` needs updating.

`LibGit2.objtype` – Function.

```
objtype(obj_type::Consts.OBJECT)
```

Return the type corresponding to the enum value.

`LibGit2.path` – Function.

```
LibGit2.path(repo::GitRepo)
```

Return the base file path of the repository `repo`.

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see `workdir` for more details).
- for bare repositories, this is the location of the "git" files.

See also gitdir, workdir.

LibGit2.peel – Function.

```
peel([T,] ref::GitReference)
```

Recursively peel **ref** until an object of type **T** is obtained. If no **T** is provided, then **ref** will be peeled until an object other than a GitTag is obtained.

- A **GitTag** will be peeled to the object it references.
- A GitCommit will be peeled to a GitTree.

  Note

  Only annotated tags can be peeled to **GitTag** objects. Lightweight tags (the default) are references under **refs/tags/** which point directly to **GitCommit** objects.

```
peel([T,] obj::GitObject)
```

Recursively peel **obj** until an object of type **T** is obtained. If no **T** is provided, then **obj** will be peeled until the type changes.

- A **GitTag** will be peeled to the object it references.
- A **GitCommit** will be peeled to a **GitTree**.

LibGit2.posixpath – Function.

```
LibGit2.posixpath(path)
```

Standardise the path string **path** to use POSIX separators.

LibGit2.push – Function.

```
push(rmt::GitRemote, refspecs; force::Bool=false, options::PushOptions=PushOptions())
```

Push to the specified **rmt** remote git repository, using **refspecs** to determine which remote branch(es) to push to. The keyword arguments are:

- **force**: if **true**, a force-push will occur, disregarding conflicts.
- **options**: determines the options for the push, e.g. which proxy headers to use. See PushOptions for more information.

Note

You can add information about the push refspecs in two other ways: by setting an option in the repository's `GitConfig` (with `push.default` as the key) or by calling `add_push!`. Otherwise you will need to explicitly specify a push refspec in the call to `push` for it to have any effect, like so: `LibGit2.push(repo, refspecs=["refs/heads/master"])`.

```
push(repo::GitRepo; kwargs...)
```

Pushes updates to an upstream of `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: the name of the upstream remote to push to.
- `remoteurl::AbstractString=""`: the URL of `remote`.
- `refspecs=AbstractString[]`: determines properties of the push.
- `force::Bool=false`: determines if the push will be a force push, overwriting the remote branch.
- `credentials=nothing`: provides credentials and/or settings when authenticating against a private `remote`.
- `callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git push [<remoteurl>¦<repo>] [<refspecs>]`.

`LibGit2.push!` – Method.

```
LibGit2.push!(w::GitRevWalker, cid::GitHash)
```

Start the `GitRevWalker` `walker` at commit `cid`. This function can be used to apply a function to all commits since a certain year, by passing the first commit of that year as `cid` and then passing the resulting `w` to `map`.

`LibGit2.push_head!` – Function.

```
LibGit2.push_head!(w::GitRevWalker)
```

Push the HEAD commit and its ancestors onto the `GitRevWalker` `w`. This ensures that HEAD and all its ancestor commits will be encountered during the walk.

`LibGit2.push_refspecs` – Function.

```
push_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the push refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to push to.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> close(remote);

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

`LibGit2.raw` – Function.

```
raw(id::GitHash) -> Vector{UInt8}
```

Obtain the raw bytes of the `GitHash` as a vector of length 20.

`LibGit2.read_tree!` – Function.

```
LibGit2.read_tree!(idx::GitIndex, tree::GitTree)
LibGit2.read_tree!(idx::GitIndex, treehash::AbstractGitHash)
```

Read the tree `tree` (or the tree pointed to by `treehash` in the repository owned by `idx`) into the index `idx`. The current index contents will be replaced.

`LibGit2.rebase!` – Function.

```
LibGit2.rebase!(repo::GitRepo, upstream::AbstractString="", newbase::AbstractString="")
```

Attempt an automatic merge rebase of the current branch, from `upstream` if provided, or otherwise from the upstream tracking branch. `newbase` is the branch to rebase onto. By default this is `upstream`.

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a `GitError`. This is roughly equivalent to the following command line statement:

```
git rebase --merge [<upstream>]
if [ -d ".git/rebase-merge" ]; then
    git rebase --abort
fi
```

LibGit2.ref_list – Function.

```
LibGit2.ref_list(repo::GitRepo) -> Vector{String}
```

Get a list of all reference names in the `repo` repository.

LibGit2.reftype – Function.

```
LibGit2.reftype(ref::GitReference) -> Cint
```

Return a `Cint` corresponding to the type of `ref`:

- `0` if the reference is invalid
- `1` if the reference is an object id
- `2` if the reference is symbolic

LibGit2.remotes – Function.

```
LibGit2.remotes(repo::GitRepo)
```

Return a vector of the names of the remotes of `repo`.

LibGit2.remove! – Function.

```
remove!(repo::GitRepo, files::AbstractString...)
remove!(idx::GitIndex, files::AbstractString...)
```

Remove all the files with paths specified by `files` in the index `idx` (or the index of the `repo`).

LibGit2.reset – Function.

```
reset(val::Integer, flag::Integer)
```

Unset the bits of `val` indexed by `flag`, returning them to `0`.

LibGit2.reset! – Function.

```
reset!(payload, [config]) -> CredentialPayload
```

Reset the `payload` state back to the initial values so that it can be used again within the credential callback. If a `config` is provided the configuration will also be updated.

Updates some entries, determined by the `pathspecs`, in the index from the target commit tree.

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

git reset [<committish>] [–] <pathspecs>...

```
reset!(repo::GitRepo, id::GitHash, mode::Cint=Consts.RESET_MIXED)
```

Reset the repository `repo` to its state at `id`, using one of three modes set by `mode`:

1. `Consts.RESET_SOFT` - move HEAD to `id`.
2. `Consts.RESET_MIXED` - default, move HEAD to `id` and reset the index to `id`.
3. `Consts.RESET_HARD` - move HEAD to `id`, reset the index to `id`, and discard all working changes.

Examples

```
# fetch changes
LibGit2.fetch(repo)
isfile(joinpath(repo_path, our_file)) # will be false

# fastforward merge the changes
LibGit2.merge!(repo, fastforward=true)

# because there was not any file locally, but there is
# a file remotely, we need to reset the branch
head_oid = LibGit2.head_oid(repo)
new_head = LibGit2.reset!(repo, head_oid, LibGit2.Consts.RESET_HARD)
```

In this example, the remote which is being fetched from does have a file called `our_file` in its index, which is why we must reset.

Equivalent to `git reset [--soft | --mixed | --hard] <id>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
head_oid = LibGit2.head_oid(repo)
```

```
open(joinpath(repo_path, "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
mode = LibGit2.Consts.RESET_HARD
# will discard the changes to file1
# and unstage it
new_head = LibGit2.reset!(repo, head_oid, mode)
```

`LibGit2.restore` – Function.

```
restore(s::State, repo::GitRepo)
```

Return a repository `repo` to a previous `State` `s`, for example the HEAD of a branch before a merge attempt. `s` can be generated using the `snapshot` function.

`LibGit2.revcount` – Function.

```
LibGit2.revcount(repo::GitRepo, commit1::AbstractString, commit2::AbstractString)
```

List the number of revisions between `commit1` and `commit2` (committish OIDs in string form). Since `commit1` and `commit2` may be on different branches, `revcount` performs a "left-right" revision list (and count), returning a tuple of `Int`s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to `git rev-list --left-right --count <commit1> <commit2>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")
println(repo_file, "hello world")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid1 = LibGit2.commit(repo, "commit 1")
println(repo_file, "hello world again")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit 2")
LibGit2.revcount(repo, string(commit_oid1), string(commit_oid2))
```

This will return `(-1, 0)`.

LibGit2.set_remote_url – Function.

```
set_remote_url(repo::GitRepo, remote_name, url)
set_remote_url(repo::String, remote_name, url)
```

Set both the fetch and push `url` for `remote_name` for the GitRepo or the git repository located at `path`. Typically git repos use `"origin"` as the remote name.

Examples

```
repo_path = joinpath(tempdir(), "Example")
repo = LibGit2.init(repo_path)
LibGit2.set_remote_url(repo, "upstream", "https://github.com/JuliaLang/Example.jl")
LibGit2.set_remote_url(repo_path, "upstream2", "https://github.com/JuliaLang/Example2.jl")
```

LibGit2.shortname – Function.

```
LibGit2.shortname(ref::GitReference)
```

Return a shortened version of the name of `ref` that's "human-readable".

```
julia> repo = LibGit2.GitRepo(path_to_repo);

julia> branch_ref = LibGit2.head(repo);

julia> LibGit2.name(branch_ref)
"refs/heads/master"

julia> LibGit2.shortname(branch_ref)
"master"
```

LibGit2.snapshot – Function.

```
snapshot(repo::GitRepo) -> State
```

Take a snapshot of the current state of the repository `repo`, storing the current HEAD, index, and any uncommitted work. The output `State` can be used later during a call to restore to return the repository to the snapshotted state.

LibGit2.split_cfg_entry – Function.

```
LibGit2.split_cfg_entry(ce::LibGit2.ConfigEntry) -> Tuple{String,String,String,String}
```

Break the `ConfigEntry` up to the following pieces: section, subsection, name, and value.

Examples

Given the git configuration file containing:

```
[credential "https://example.com"]
    username = me
```

The `ConfigEntry` would look like the following:

```
julia> entry
ConfigEntry("credential.https://example.com.username", "me")

julia> LibGit2.split_cfg_entry(entry)
("credential", "https://example.com", "username", "me")
```

Refer to the git config syntax documentation for more details.

`LibGit2.status` – Function.

```
LibGit2.status(repo::GitRepo, path::String) -> Union{Cuint, Cvoid}
```

Lookup the status of the file at `path` in the git repository `repo`. For instance, this can be used to check if the file at `path` has been modified and needs to be staged and committed.

`LibGit2.stage` – Function.

```
stage(ie::IndexEntry) -> Cint
```

Get the stage number of `ie`. The stage number `0` represents the current state of the working tree, but other numbers can be used in the case of a merge conflict. In such a case, the various stage numbers on an `IndexEntry` describe which side(s) of the conflict the current state of the file belongs to. Stage `0` is the state before the attempted merge, stage `1` is the changes which have been made locally, stages `2` and larger are for changes from other branches (for instance, in the case of a multi-branch "octopus" merge, stages `2`, `3`, and `4` might be used).

`LibGit2.tag_create` – Function.

```
LibGit2.tag_create(repo::GitRepo, tag::AbstractString, commit; kwargs...)
```

Create a new git tag `tag` (e.g. `"v0.5"`) in the repository `repo`, at the commit `commit`.

The keyword arguments are:

- `msg::AbstractString=""`: the message for the tag.
- `force::Bool=false`: if `true`, existing references will be overwritten.
- `sig::Signature=Signature(repo)`: the tagger's signature.

`LibGit2.tag_delete` – Function.

```
LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

`LibGit2.tag_list` – Function.

```
LibGit2.tag_list(repo::GitRepo) -> Vector{String}
```

Get a list of all tags in the git repository `repo`.

`LibGit2.target` – Function.

```
LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

`LibGit2.toggle` – Function.

```
toggle(val::Integer, flag::Integer)
```

Flip the bits of `val` indexed by `flag`, so that if a bit is `0` it will be `1` after the toggle, and vice-versa.

`LibGit2.transact` – Function.

```
transact(f::Function, repo::GitRepo)
```

Apply function `f` to the git repository `repo`, taking a `snapshot` before applying `f`. If an error occurs within `f`, `repo` will be returned to its snapshot state using `restore`. The error which occurred will be rethrown, but the state of `repo` will not be corrupted.

`LibGit2.treewalk` – Function.

```
treewalk(f, tree::GitTree, post::Bool=false)
```

Traverse the entries in `tree` and its subtrees in post or pre order. Preorder means beginning at the root and then traversing the leftmost subtree (and recursively on down through that subtree's leftmost subtrees) and moving right through the subtrees. Postorder means beginning at the bottom of the leftmost subtree, traversing upwards through it, then traversing the next right subtree (again beginning at the bottom) and finally visiting the tree root last of all.

The function parameter `f` should have following signature:

```
(String, GitTreeEntry) -> Cint
```

A negative value returned from `f` stops the tree walk. A positive value means that the entry will be skipped if `post` is `false`.

`LibGit2.upstream` – Function.

```
upstream(ref::GitReference) -> Union{GitReference, Nothing}
```

Determine if the branch containing `ref` has a specified upstream branch.

Return either a `GitReference` to the upstream branch if it exists, or `nothing` if the requested branch does not have an upstream counterpart.

`LibGit2.update!` – Function.

```
update!(repo::GitRepo, files::AbstractString...)
update!(idx::GitIndex, files::AbstractString...)
```

Update all the files with paths specified by `files` in the index `idx` (or the index of the `repo`). Match the state of each file in the index with the current state on disk, removing it if it has been removed on disk, or updating its entry in the object database.

`LibGit2.url` – Function.

```
url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.init(mktempdir());
```

```
julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> LibGit2.url(remote)
"https://github.com/JuliaLang/Example.jl"
```

LibGit2.version – Function.

```
version() -> VersionNumber
```

Return the version of libgit2 in use, as a VersionNumber.

LibGit2.with – Function.

```
with(f::Function, obj)
```

Resource management helper function. Applies `f` to `obj`, making sure to call `close` on `obj` after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed.

LibGit2.with_warn – Function.

```
with_warn(f::Function, ::Type{T}, args...)
```

Resource management helper function. Apply `f` to `args`, first constructing an instance of type `T` from `args`. Makes sure to call `close` on the resulting object after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed. If an error is thrown by `f`, a warning is shown containing the error.

LibGit2.workdir – Function.

```
LibGit2.workdir(repo::GitRepo)
```

Return the location of the working directory of `repo`. This will throw an error for bare repositories.

> Note
>
> This will typically be the parent directory of `gitdir(repo)`, but can be different in some cases: e.g. if either the `core.worktree` configuration variable or the `GIT_WORK_TREE` environment variable is set.

See also gitdir, path.

LibGit2.GitObject – Method.

```
(::Type{T})(te::GitTreeEntry) where T<:GitObject
```

Get the git object to which `te` refers and return it as its actual type (the type `entrytype` would show), for instance a `GitBlob` or `GitTag`.

Examples

```
tree = LibGit2.GitTree(repo, "HEAD^{tree}")
tree_entry = tree[1]
blob = LibGit2.GitBlob(tree_entry)
```

`LibGit2.UserPasswordCredential` – Type.

Credential that support only `user` and `password` parameters

`LibGit2.SSHCredential` – Type.

SSH credential type

`LibGit2.isfilled` – Function.

```
isfilled(cred::AbstractCredential) -> Bool
```

Verifies that a credential is ready for use in authentication.

`LibGit2.CachedCredentials` – Type.

Caches credential information for re-use

`LibGit2.CredentialPayload` – Type.

```
LibGit2.CredentialPayload
```

Retains the state between multiple calls to the credential callback for the same URL. A `CredentialPayload` instance is expected to be `reset!` whenever it will be used with a different URL.

`LibGit2.approve` – Function.

```
approve(payload::CredentialPayload; shred::Bool=true) -> Nothing
```

Store the `payload` credential for re-use in a future authentication. Should only be called when authentication was successful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

`LibGit2.reject` — Function.

```
reject(payload::CredentialPayload; shred::Bool=true) -> Nothing
```

Discard the `payload` credential from begin re-used in future authentication. Should only be called when authentication was unsuccessful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

Chapter 76

Dynamic Linker

Libdl.dlopen – Function.

```
dlopen(libfile::AbstractString [, flags::Integer]; throw_error:Bool = true)
```

Load a shared library, returning an opaque handle.

The extension given by the constant `dlext` (`.so`, `.dll`, or `.dylib`) can be omitted from the `libfile` string, as it is automatically appended if needed. If `libfile` is not an absolute path name, then the paths in the array `DL_LOAD_PATH` are searched for `libfile`, followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of `RTLD_LOCAL`, `RTLD_GLOBAL`, `RTLD_LAZY`, `RTLD_NOW`, `RTLD_NODELETE`, `RTLD_NOLOAD`, `RTLD_DEEPBIND`, and `RTLD_FIRST`. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default `dlopen` flags are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` while on other platforms the defaults are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL`. An important usage of these flags is to specify non default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

If the library cannot be found, this method throws an error, unless the keyword argument `throw_error` is set to `false`, in which case this method returns `nothing`.

Libdl.dlopen_e – Function.

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to dlopen, except returns C_NULL instead of raising errors. This method is now deprecated in favor of dlopen(libfile::AbstractString [, flags::Integer]; throw_error=false).

Libdl.RTLD_NOW – Constant.

```
RTLD_DEEPBIND
RTLD_FIRST
RTLD_GLOBAL
RTLD_LAZY
RTLD_LOCAL
RTLD_NODELETE
RTLD_NOLOAD
RTLD_NOW
```

Enum constant for dlopen. See your platform man page for details, if applicable.

Libdl.dlsym – Function.

```
dlsym(handle, sym)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

Libdl.dlsym_e – Function.

```
dlsym_e(handle, sym)
```

Look up a symbol from a shared library handle, silently return C_NULL on lookup failure. This method is now deprecated in favor of dlsym(handle, sym; throw_error=false).

Libdl.dlclose – Function.

```
dlclose(handle)
```

Close shared library referenced by handle.

```
dlclose(::Nothing)
```

For the very common pattern usage pattern of

```
try
    hdl = dlopen(library_name)
    ... do something
finally
    dlclose(hdl)
end
```

We define a `dlclose()` method that accepts a parameter of type `Nothing`, so that user code does not have to change its behavior for the case that `library_name` was not found.

`Libdl.dlext` – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

`Libdl.find_library` – Function.

```
find_library(names, locations)
```

Searches for the first library in `names` in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be dlopen'd. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a `global const` and used as the library name in future `ccall`'s. On failure, it returns the empty string.

`Base.DL_LOAD_PATH` – Constant.

```
DL_LOAD_PATH
```

When calling `dlopen`, the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

Chapter 77

Linear Algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with `using LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1

julia> tr(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Array{Float64,2}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

As well as other useful operations, such as finding eigenvalues or eigenvectors:

```
julia> A = [-4. -17.; 2. 2.]
2×2 Array{Float64,2}:
```

```
 -4.0  -17.0
  2.0    2.0

julia> eigvals(A)
2-element Array{Complex{Float64},1}:
 -1.0 - 5.0im
 -1.0 + 5.0im

julia> eigvecs(A)
2×2 Array{Complex{Float64},2}:
  0.945905-0.0im        0.945905+0.0im
 -0.166924+0.278207im  -0.166924-0.278207im
```

In addition, Julia provides many factorizations which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. See the documentation on `factorize` for more information. As an example:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
  1.0    0.0       0.0
 -0.15   1.0       0.0
 -0.3   -0.132196  1.0
U factor:
3×3 Array{Float64,2}:
 -10.0  2.3     4.0
   0.0  2.345  -3.4
   0.0  0.0    -5.24947
```

Since `A` is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
3×3 Tridiagonal{Float64,Array{Float64,1}}:
 -1.64286   0.0   ⋅
  0.0      -2.8  0.0
   ⋅        0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.142857  -0.8
  ⋅   1.0       -0.6
  ⋅    ⋅         1.0
permutation:
3-element Array{Int64,1}:
 1
 2
 3
```

Here, Julia was able to detect that `B` is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for a matrix that is known to have certain properties e.g. it is symmetric, or tridiagonal. Julia provides some special types so that you can "tag" matrices as having these properties. For instance:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

sB has been tagged as a matrix that's (real) symmetric, so for later operations we might perform on it, such as eigenfactorization or computing matrix-vector products, efficiencies can be found by only referencing half of it. For example:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> x = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> sB\x
3-element Array{Float64,1}:
 -1.7391304347826084
 -1.1086956521739126
 -1.4565217391304346
```

The \ operation here performs the linear solution. The left-division operator is pretty powerful and it's easy to write compact, readable code that is flexible enough to solve all sorts of systems of linear equations.

77.1 Special matrices

Matrices with special symmetries and structures arise often in linear algebra and are frequently associated with various matrix factorizations. Julia features a rich collection of special matrix types, which allow for fast computation with specialized routines that are specially developed for particular matrix types.

The following tables summarize the types of special matrices that have been implemented in Julia, as well as whether hooks to various optimized methods for them in LAPACK are available.

Type	Description
`Symmetric`	Symmetric matrix
`Hermitian`	Hermitian matrix
`UpperTriangular`	Upper triangular matrix
`UnitUpperTriangular`	Upper triangular matrix with unit diagonal
`LowerTriangular`	Lower triangular matrix
`UnitLowerTriangular`	Lower triangular matrix with unit diagonal
`UpperHessenberg`	Upper Hessenberg matrix
`Tridiagonal`	Tridiagonal matrix
`SymTridiagonal`	Symmetric tridiagonal matrix
`Bidiagonal`	Upper/lower bidiagonal matrix
`Diagonal`	Diagonal matrix
`UniformScaling`	Uniform scaling operator

Elementary operations

Matrix type	+	-	*	\	Other functions with optimized methods
`Symmetric`				MV	`inv`, `sqrt`, `exp`
`Hermitian`				MV	`inv`, `sqrt`, `exp`
`UpperTriangular`			MV	MV	`inv`, `det`
`UnitUpperTriangular`			MV	MV	`inv`, `det`
`LowerTriangular`			MV	MV	`inv`, `det`
`UnitLowerTriangular`			MV	MV	`inv`, `det`
`UpperHessenberg`				MM	`inv`, `det`
`SymTridiagonal`	M	M	MS	MV	`eigmax`, `eigmin`
`Tridiagonal`	M	M	MS	MV	
`Bidiagonal`	M	M	MS	MV	
`Diagonal`	M	M	MV	MV	`inv`, `det`, `logdet`, `/`
`UniformScaling`	M	M	MVS	MVS	`/`

Legend:

Key	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Matrix type	LAPACK	eigen	eigvals	eigvecs	svd	svdvals
Symmetric	SY		ARI			
Hermitian	HE		ARI			
UpperTriangular	TR	A	A	A		
UnitUpperTriangular	TR	A	A	A		
LowerTriangular	TR	A	A	A		
UnitLowerTriangular	TR	A	A	A		
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

Matrix factorizations

Legend:

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. `eigvals(M)`
R (range)	An optimized method to find the `il`th through the `ih`th characteristic values are available	`eigvals(M, il, ih)`
I (interval)	An optimized method to find the characteristic values in the interval `[vl, vh]` is available	`eigvals(M, vl, vh)`
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values `x=[x1, x2,...]` is available	`eigvecs(M, x)`

The uniform scaling operator

A `UniformScaling` operator represents a scalar times the identity operator, `λ*I`. The identity operator `I` is defined as a constant and is an instance of `UniformScaling`. The size of these operators are generic and match the other matrix in the binary operations `+`, `-`, `*` and `\`. For `A+I` and `A-I` this means that `A` must be square. Multiplication with the identity

operator `I` is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

To see the `UniformScaling` operator in action:

```
julia> U = UniformScaling(2);

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> a + U
2×2 Array{Int64,2}:
 3  2
 3  6

julia> a * U
2×2 Array{Int64,2}:
 2  4
 6  8

julia> [a U]
2×4 Array{Int64,2}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> b - U
ERROR: DimensionMismatch("matrix is not square: dimensions are (2, 3)")
Stacktrace:
[...]
```

If you need to solve many systems of the form `(A+μI)x = b` for the same `A` and different `μ`, it might be beneficial to first compute the Hessenberg factorization `F` of `A` via the `hessenberg` function. Given `F`, Julia employs an efficient algorithm for `(F+μ*I) \ b` (equivalent to `(A+μ*I)x \ b`) and related operations like determinants.

77.2 Matrix factorizations

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the Standard Functions section of the Linear Algebra documentation.

Type	Description
`BunchKaufman`	Bunch-Kaufman factorization
`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LDLt`	LDL(T) factorization
`LU`	LU factorization
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`LQ`	QR factorization of `transpose(A)`
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`GeneralizedEigen`	Generalized spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD
`Schur`	Schur decomposition
`GeneralizedSchur`	Generalized Schur decomposition

77.3 Standard Functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

`Base.:*` — Method.

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Array{Int64,2}:
 2  1
 1  1
```

Base.:\ – Method.

```
\(A, B)
```

Matrix division using a polyalgorithm. For input matrices `A` and `B`, the result `X` is such that `A*X == B` when `A` is square. The solver that is used depends upon the structure of `A`. If `A` is upper or lower triangular (or diagonal), no factorization of `A` is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.

For rectangular `A` the result is the minimum-norm least squares solution computed by a pivoted QR factorization of `A` and a rank estimate of `A` based on the R factor.

When `A` is sparse, a similar polyalgorithm is used. For indefinite matrices, the `LDLt` factorization does not use pivoting during the numerical factorization and therefore the procedure can fail even for invertible matrices.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];

julia> X = A \ B
2-element Array{Float64,1}:
 32.0
 18.0

julia> A * X == B
true
```

LinearAlgebra.SingularException – Type.

```
SingularException
```

Exception thrown when the input matrix has one or more zero-valued eigenvalues, and is not invertible. A linear solve involving such a matrix cannot be computed. The `info` field indicates the location of (one of) the singular value(s).

LinearAlgebra.PosDefException – Type.

```
PosDefException
```

Exception thrown when the input matrix was not positive definite. Some linear algebra functions and factorizations are only applicable to positive definite matrices. The `info` field indicates the location of (one of) the eigenvalue(s) which is (are) less than/equal to 0.

LinearAlgebra.dot – Function.

```
dot(x, y)
x ⋅ y
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated.

`dot` also works on arbitrary iterable objects, including arrays of any dimension, as long as `dot` is defined on the elements.

`dot` is semantically equivalent to `sum(dot(vx,vy) for (vx,vy) in zip(x, y))`, with the added restriction that the arguments must have equal lengths.

`x ⋅ y` (where `⋅` can be typed by tab-completing `\cdot` in the REPL) is a synonym for `dot(x, y)`.

Examples

```
julia> dot([1; 1], [2; 3])
5

julia> dot([im; im], [1; 1])
0 - 2im

julia> dot(1:5, 2:6)
70

julia> x = fill(2., (5,5));

julia> y = fill(3., (5,5));

julia> dot(x, y)
150.0
```

LinearAlgebra.cross – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]
3-element Array{Int64,1}:
 0
 1
 0

julia> b = [0;0;1]
3-element Array{Int64,1}:
 0
 0
 1

julia> cross(a,b)
3-element Array{Int64,1}:
 1
 0
 0
```

LinearAlgebra.factorize – Function.

```
factorize(A)
```

Compute a convenient factorization of `A`, based upon the type of the input matrix. `factorize` checks `A` to see if it is symmetric/triangular/etc. if `A` is passed as a generic matrix. `factorize` checks every element of `A` to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C`.

If `factorize` is called on a Hermitian positive-definite matrix, for instance, then `factorize` will return a Cholesky factorization.

Examples

```
julia> A = Array(Bidiagonal(fill(1.0, (5, 5)), :U))
5×5 Array{Float64,2}:
```

Properties of `A`	type of factorization
Positive-definite	Cholesky (see `cholesky`)
Dense Symmetric/Hermitian	Bunch-Kaufman (see `bunchkaufman`)
Sparse Symmetric/Hermitian	LDLt (see `ldlt`)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see `lu`)
Symmetric real tridiagonal	LDLt (see `ldlt`)
General square	LU (see `lu`)
General non-square	QR (see `qr`)

```
 1.0  1.0  0.0  0.0  0.0
 0.0  1.0  1.0  0.0  0.0
 0.0  0.0  1.0  1.0  0.0
 0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0

julia> factorize(A) # factorize will check to see that A is already factorized
5×5 Bidiagonal{Float64,Array{Float64,1}}:
 1.0  1.0   ·    ·    ·
  ·   1.0  1.0   ·    ·
  ·    ·   1.0  1.0   ·
  ·    ·    ·   1.0  1.0
  ·    ·    ·    ·   1.0
```

This returns a `5×5 Bidiagonal{Float64}`, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for `Bidiagonal` types.

`LinearAlgebra.Diagonal` — Type.

```
Diagonal(A::AbstractMatrix)
```

Construct a matrix from the diagonal of `A`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> Diagonal(A)
3×3 Diagonal{Int64,Array{Int64,1}}:
 1  ·  ·
 ·  5  ·
 ·  ·  9
```

```
Diagonal(V::AbstractVector)
```

Construct a matrix with `V` as its diagonal.

Examples

```
julia> V = [1, 2]
2-element Array{Int64,1}:
 1
 2

julia> Diagonal(V)
2×2 Diagonal{Int64,Array{Int64,1}}:
 1  ·
 ·  2
```

`LinearAlgebra.Bidiagonal` — Type.

```
Bidiagonal(dv::V, ev::V, uplo::Symbol) where V <: AbstractVector
```

Constructs an upper (`uplo=:U`) or lower (`uplo=:L`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The length of `ev` must be one less than the length of `dv`.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
```

```
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first superdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  7  ⋅  ⋅
 ⋅  2  8  ⋅
 ⋅  ⋅  3  9
 ⋅  ⋅  ⋅  4

julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  ⋅  ⋅  ⋅
 7  2  ⋅  ⋅
 ⋅  8  3  ⋅
 ⋅  ⋅  9  4
```

```
Bidiagonal(A, uplo::Symbol)
```

Construct a `Bidiagonal` matrix from the main diagonal of `A` and its first super- (if `uplo=:U`) or sub-diagonal (if `uplo=:L`).

Examples

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Array{Int64,2}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, :U) # contains the main diagonal and first superdiagonal of A
4×4 Bidiagonal{Int64,Array{Int64,1}}:
```

```
 1  1  ·  ·
 ·  2  2  ·
 ·  ·  3  3
 ·  ·  ·  4

julia> Bidiagonal(A, :L) # contains the main diagonal and first subdiagonal of A
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  ·  ·  ·
 2  2  ·  ·
 ·  3  3  ·
 ·  ·  4  4
```

LinearAlgebra.SymTridiagonal – Type.

```
SymTridiagonal(dv::V, ev::V) where V <: AbstractVector
```

Construct a symmetric tridiagonal matrix from the diagonal (`dv`) and first sub/super-diagonal (`ev`), respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

For `SymTridiagonal` block matrices, the elements of `dv` are symmetrized. The argument `ev` is interpreted as the superdiagonal. Blocks from the subdiagonal are (materialized) transpose of the corresponding superdiagonal blocks.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64,Array{Int64,1}}:
```

```
 1  7  ·  ·
 7  2  8  ·
 ·  8  3  9
 ·  ·  9  4

julia> A = SymTridiagonal(fill([1 2; 3 4], 3), fill([1 2; 3 4], 2));

julia> A[1,1]
2×2 Symmetric{Int64,Array{Int64,2}}:
 1  2
 2  4

julia> A[1,2]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A[2,1]
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first superdiagonal of the symmetric matrix `A`.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Array{Int64,2}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64,Array{Int64,1}}:
 1  2  ·
 2  4  5
 ·  5  6

julia> B = reshape([[1 2; 2 3], [1 2; 3 4], [1 3; 2 4], [1 2; 2 3]], 2, 2);
```

```
julia> SymTridiagonal(B)
2×2 SymTridiagonal{Array{Int64,2},Array{Array{Int64,2},1}}:
 [1 2; 2 3]  [1 3; 2 4]
 [1 2; 3 4]  [1 2; 2 3]
```

LinearAlgebra.Tridiagonal — Type.

```
Tridiagonal(dl::V, d::V, du::V) where V <: AbstractVector
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Examples

```
julia> dl = [1, 2, 3];

julia> du = [4, 5, 6];

julia> d = [7, 8, 9, 0];

julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 7  4  ·  ·
 1  8  5  ·
 ·  2  9  6
 ·  ·  3  0
```

```
Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix `A`.

Examples

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Array{Int64,2}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
```

```
 1  2  3  4


julia> Tridiagonal(A)
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 1  2  ·  ·
 1  2  3  ·
 ·  2  3  4
 ·  ·  3  4
```

LinearAlgebra.Symmetric – Type.

```
Symmetric(A, uplo=:U)
```

Construct a `Symmetric` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Examples

```
julia> A = [1 0 2 0 3; 0 4 0 5 0; 6 0 7 0 8; 0 9 0 1 0; 2 0 3 0 4]
5×5 Array{Int64,2}:
 1  0  2  0  3
 0  4  0  5  0
 6  0  7  0  8
 0  9  0  1  0
 2  0  3  0  4


julia> Supper = Symmetric(A)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  2  0  3
 0  4  0  5  0
 2  0  7  0  8
 0  5  0  1  0
 3  0  8  0  4


julia> Slower = Symmetric(A, :L)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  6  0  2
 0  4  0  9  0
 6  0  7  0  3
 0  9  0  1  0
 2  0  3  0  4
```

Note that `Supper` will not be equal to `Slower` unless `A` is itself symmetric (e.g. if `A == transpose(A)`).

`LinearAlgebra.Hermitian` — Type.

```
Hermitian(A, uplo=:U)
```

Construct a `Hermitian` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Examples

```
julia> A = [1 0 2+2im 0 3-3im; 0 4 0 5 0; 6-6im 0 7 0 8+8im; 0 9 0 1 0; 2+2im 0 3-3im 0 4];

julia> Hupper = Hermitian(A)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  2+2im  0+0im  3-3im
 0+0im  4+0im  0+0im  5+0im  0+0im
 2-2im  0+0im  7+0im  0+0im  8+8im
 0+0im  5+0im  0+0im  1+0im  0+0im
 3+3im  0+0im  8-8im  0+0im  4+0im

julia> Hlower = Hermitian(A, :L)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  6+6im  0+0im  2-2im
 0+0im  4+0im  0+0im  9+0im  0+0im
 6-6im  0+0im  7+0im  0+0im  3+3im
 0+0im  9+0im  0+0im  1+0im  0+0im
 2+2im  0+0im  3-3im  0+0im  4+0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if `A == adjoint(A)`).

All non-real parts of the diagonal will be ignored.

```
Hermitian(fill(complex(1,1), 1, 1)) == fill(1, 1, 1)
```

`LinearAlgebra.LowerTriangular` — Type.

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> LowerTriangular(A)
3×3 LowerTriangular{Float64,Array{Float64,2}}:
 1.0   ·    ·
 4.0  5.0   ·
 7.0  8.0  9.0
```

LinearAlgebra.UpperTriangular – Type.

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UpperTriangular(A)
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0  3.0
  ·   5.0  6.0
  ·    ·   9.0
```

LinearAlgebra.UnitLowerTriangular – Type.

```
UnitLowerTriangular(A::AbstractMatrix)
```

Construct a `UnitLowerTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitLowerTriangular(A)
3×3 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0   ·    ·
 4.0  1.0   ·
 7.0  8.0  1.0
```

LinearAlgebra.UnitUpperTriangular — Type.

```
UnitUpperTriangular(A::AbstractMatrix)
```

Construct an `UnitUpperTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitUpperTriangular(A)
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0  3.0
  ·   1.0  6.0
  ·    ·   1.0
```

LinearAlgebra.UpperHessenberg — Type.

```
UpperHessenberg(A::AbstractMatrix)
```

Construct an `UpperHessenberg` view of the matrix `A`. Entries of `A` below the first subdiagonal are ignored.

Efficient algorithms are implemented for `H \ b`, `det(H)`, and similar.

See also the `hessenberg` function to factor any matrix into a similar upper-Hessenberg matrix.

If `F::Hessenberg` is the factorization object, the unitary matrix can be accessed with `F.Q` and the Hessenberg matrix with `F.H`. When `Q` is extracted, the resulting type is the `HessenbergQ` object, and may be converted to a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Iterating the decomposition produces the factors `F.Q` and `F.H`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
4×4 Array{Int64,2}:
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16

julia> UpperHessenberg(A)
4×4 UpperHessenberg{Int64,Array{Int64,2}}:
 1   2   3   4
 5   6   7   8
 ·  10  11  12
 ·   ·  15  16
```

`LinearAlgebra.UniformScaling` — Type.

```
UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator, `λ*I`. See also `I`.

Examples

```
julia> J = UniformScaling(2.)
UniformScaling{Float64}
2.0*I

julia> A = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> J*A
2×2 Array{Float64,2}:
```

```
2.0  4.0
6.0  8.0
```

LinearAlgebra.Factorization – Type.

```
LinearAlgebra.Factorization
```

Abstract type for matrix factorizations a.k.a. matrix decompositions. See online documentation for a list of available matrix factorizations.

LinearAlgebra.LU – Type.

```
LU <: Factorization
```

Matrix factorization type of the `LU` factorization of a square matrix `A`. This is the return type of `lu`, the corresponding matrix factorization function.

The individual components of the factorization `F::LU` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (unit lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

Examples

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
```

```
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu` — Function.

```
lu(A, pivot=Val(true); check = true) -> F::LU
```

Compute the LU factorization of `A`.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

In most cases, if `A` is a subtype `S` of `AbstractMatrix{T}` with an element type `T` supporting `+`, `-`, `*` and `/`, the return type is `LU{T,S{T}}`. If pivoting is chosen (default) the element type should also support `abs` and `<`.

The individual components of the factorization `F` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

The relationship between `F` and `A` is

`F.L*F.U == A[F.p, :]`

`F` further supports the following functions:

Examples

Supported function	LU	LU{T,Tridiagonal{T}}
/	☒	
\	☒	☒
inv	☒	☒
det	☒	☒
logdet	☒	☒
logabsdet	☒	☒
size	☒	☒

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

LinearAlgebra.lu! – Function.

```
lu!(A, pivot=Val(true); check = true) -> LU
```

`lu!` is the same as lu, but saves space by overwriting the input `A`, instead of creating a copy. An InexactError

exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Array{Float64,2}:
 4.0  3.0
 6.0  3.0

julia> F = lu!(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> iA = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> lu!(iA)
ERROR: InexactError: Int64(0.6666666666666666)
Stacktrace:
[...]
```

`LinearAlgebra.Cholesky` — Type.

```
Cholesky <: Factorization
```

Matrix factorization type of the Cholesky factorization of a dense symmetric/Hermitian positive definite matrix `A`. This is the return type of `cholesky`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::Cholesky` via `F.L` and `F.U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

LinearAlgebra.CholeskyPivoted – Type.

```
CholeskyPivoted
```

Matrix factorization type of the pivoted Cholesky factorization of a dense symmetric/Hermitian positive semi-definite matrix `A`. This is the return type of `cholesky(_, Val(true))`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::CholeskyPivoted` via `F.L` and `F.U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A, Val(true))
CholeskyPivoted{Float64,Array{Float64,2}}
U factor with rank 3:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 9.89949  -4.34366  -1.61624
  ·        4.25825   1.1694
  ·         ·        0.142334
permutation:
3-element Array{Int64,1}:
 3
 2
 1
```

LinearAlgebra.cholesky – Function.

```
cholesky(A, Val(false); check = true) -> Cholesky
```

Compute the Cholesky factorization of a dense symmetric positive definite matrix `A` and return a `Cholesky` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian StridedMatrix` or a perfectly symmetric or Hermitian `StridedMatrix`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F.L` and `F.U`. The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0
```

```
julia> C = cholesky(A)
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

```
cholesky(A, Val(true); tol = 0.0, check = true) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix `A` and return a `CholeskyPivoted` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian StridedMatrix` or a perfectly symmetric or Hermitian `StridedMatrix`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F.L` and `F.U`. The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`. The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

`LinearAlgebra.cholesky!` – Function.

```
cholesky!(A, Val(false); check = true) -> Cholesky
```

The same as cholesky, but saves space by overwriting the input A, instead of creating a copy. An InexactError exception is thrown if the factorization produces a number not representable by the element type of A, e.g. for integer types.

Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1   2
 2  50

julia> cholesky!(A)
ERROR: InexactError: Int64(6.782329983125268)
Stacktrace:
[...]
```

```
cholesky!(A, Val(true); tol = 0.0, check = true) -> CholeskyPivoted
```

The same as cholesky, but saves space by overwriting the input A, instead of creating a copy. An InexactError exception is thrown if the factorization produces a number not representable by the element type of A, e.g. for integer types.

LinearAlgebra.lowrankupdate – Function.

```
lowrankupdate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U + v*v') but the computation of CC only uses O(n^2) operations.

LinearAlgebra.lowrankdowndate – Function.

```
lowrankdowndate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U - v*v') but the computation of CC only uses O(n^2) operations.

LinearAlgebra.lowrankupdate! – Function.

```
lowrankupdate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U + v*v')` but the computation of `CC` only uses `O(n^2)` operations. The input factorization `C` is updated in place such that on exit `C == CC`. The vector `v` is destroyed during the computation.

`LinearAlgebra.lowrankdowndate!` – Function.

```
lowrankdowndate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U - v*v')` but the computation of `CC` only uses `O(n^2)` operations. The input factorization `C` is updated in place such that on exit `C == CC`. The vector `v` is destroyed during the computation.

`LinearAlgebra.LDLt` – Type.

```
LDLt <: Factorization
```

Matrix factorization type of the `LDLt` factorization of a real `SymTridiagonal` matrix `S` such that `S = L*Diagonal(d)*L'`, where `L` is a `UnitLowerTriangular` matrix and `d` is a vector. The main use of an `LDLt` factorization `F = ldlt(S)` is to solve the linear system of equations `Sx = b` with `F\b`. This is the return type of `ldlt`, the corresponding matrix factorization function.

The individual components of the factorization `F::LDLt` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (unit lower triangular) part of `LDLt`
`F.D`	`D` (diagonal) part of `LDLt`
`F.Lt`	`Lt` (unit upper triangular) part of `LDLt`
`F.d`	diagonal values of `D` as a `Vector`

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0

julia> F = ldlt(S)
LDLt{Float64,SymTridiagonal{Float64,Array{Float64,1}}}
L factor:
```

```
3×3 UnitLowerTriangular{Float64,SymTridiagonal{Float64,Array{Float64,1}}}:
 1.0        ·          ·
 0.333333  1.0         ·
 0.0       0.545455   1.0
D factor:
3×3 Diagonal{Float64,Array{Float64,1}}:
 3.0   ·        ·
  ·   3.66667   ·
  ·    ·       3.90909
```

`LinearAlgebra.ldlt` – Function.

```
ldlt(S::SymTridiagonal) -> LDLt
```

Compute an `LDLt` factorization of the real symmetric tridiagonal matrix `S` such that `S = L*Diagonal(d)*L'` where `L` is a unit lower triangular matrix and `d` is a vector. The main use of an `LDLt` factorization `F = ldlt(S)` is to solve the linear system of equations `Sx = b` with `F\b`.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0

julia> ldltS = ldlt(S);

julia> b = [6., 7., 8.];

julia> ldltS \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255

julia> S \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

LinearAlgebra.ldlt! – Function.

```
ldlt!(S::SymTridiagonal) -> LDLt
```

Same as ldlt, but saves space by overwriting the input S, instead of creating a copy.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0

julia> ldltS = ldlt!(S);

julia> ldltS === S
false

julia> S
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0       0.333333   ·
 0.333333  3.66667   0.545455
  ·        0.545455  3.90909
```

LinearAlgebra.QR – Type.

```
QR <: Factorization
```

A QR matrix factorization stored in a packed format, typically obtained from qr. If A is an m×n matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components Q and R.

The object has two fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.

LinearAlgebra.QRCompactWY – Type.

```
QRCompactWY <: Factorization
```

A QR matrix factorization stored in a compact blocked format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the `QR` format except that the orthogonal/unitary matrix Q is stored in Compact WY format [1]. For the block size n_b, it is stored as a `m×n` lower trapezoidal matrix V and a matrix $T = (T_1\ T_2\ ...\ T_{b-1}\ T_b')$ composed of $b = \lceil \mathsf{min}(m,n)/n_b \rceil$ upper triangular matrices T_j of size $n_b \times n_b$ ($j = 1, ..., b-1$) and an upper trapezoidal $n_b \times \mathsf{min}(m,n) - (b-1)n_b$ matrix T_b' ($j = b$) whose upper square part denoted with T_b satisfying

$$Q = \prod_{i=1}^{\mathsf{min}(m,n)} (I - \tau_i v_i v_i^T) = \prod_{j=1}^{b} (I - V_j T_j V_j^T)$$

such that v_i is the ith column of V, τ_i is the ith element of `[diag(T_1); diag(T_2); …; diag(T_b)]`, and $(V_1\ V_2\ ...\ V_b)$ is the left `m×min(m, n)` block of V. When constructed using `qr`, the block size is given by $n_b = \mathsf{min}(m, n, 36)$.

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors`, as in the `QR` type, is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format such that `V = I + tril(F.factors, -1)`.
- `T` is a n_b-by-$\mathsf{min}(m, n)$ matrix as described above. The subdiagonal elements for each triangular matrix T_j are ignored.

Note

This format should not to be confused with the older WY representation [2].

LinearAlgebra.QRPivoted – Type.

```
QRPivoted <: Factorization
```

A QR matrix factorization with column pivoting in a packed format, typically obtained from qr. If A is an `m×n` matrix, then

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q`, `R`, and `p`.

The object has three fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.
- `jpvt` is an integer vector of length `n` corresponding to the permutation P.

LinearAlgebra.qr – Function.

```
qr(A, pivot=Val(false); blocksize) -> F
```

[2] C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009

[1] R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

$$A = QR$$

The returned object `F` stores the factorization in a packed format:

- if `pivot == Val(true)` then `F` is a `QRPivoted` object,
- otherwise if the element type of `A` is a BLAS type (`Float32`, `Float64`, `ComplexF32` or `ComplexF64`), then `F` is a `QRCompactWY` object,
- otherwise `F` is a `QR` object.

The individual components of the decomposition `F` can be retrieved via property accessors:

- `F.Q`: the orthogonal/unitary matrix `Q`
- `F.R`: the upper triangular matrix `R`
- `F.p`: the permutation vector of the pivot (`QRPivoted` only)
- `F.P`: the permutation matrix of the pivot (`QRPivoted` only)

Iterating the decomposition produces the components `Q`, `R`, and if extant `p`.

The following functions are available for the `QR` objects: `inv`, `size`, and `\`. When `A` is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned. When `A` is not full rank, factorization with (column) pivoting is required to obtain a minimum norm solution.

Multiplication with respect to either full/square or non-full/square `Q` is allowed, i.e. both `F.Q*F.R` and `F.Q*A` are supported. A `Q` matrix can be converted into a regular matrix with `Matrix`. This operation returns the "thin" Q factor, i.e., if `A` is `m×n` with `m>=n`, then `Matrix(F.Q)` yields an `m×n` matrix with orthonormal columns. To retrieve the "full" Q factor, an `m×m` orthogonal matrix, use `F.Q*Matrix(I,m,m)`. If `m<=n`, then `Matrix(F.Q)` yields an `m×m` orthogonal matrix.

The block size for QR decomposition can be specified by keyword argument `blocksize :: Integer` when `pivot == Val(false)` and `A isa StridedMatrix{<:BlasFloat}`. It is ignored when `blocksize > minimum(size(A))`. See `QRCompactWY`.

> Julia 1.4
>
> The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Array{Float64,2}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qr(A)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
3×3 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.6   0.0   0.8
 -0.8   0.0  -0.6
  0.0  -1.0   0.0
R factor:
2×2 Array{Float64,2}:
 -5.0  10.0
  0.0  -1.0

julia> F.Q * F.R == A
true
```

Note

`qr` returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the `Q` and `R` matrices can be stored compactly rather as two separate dense matrices.

`LinearAlgebra.qr!` – Function.

```
qr!(A, pivot=Val(false); blocksize)
```

`qr!` is the same as `qr` when `A` is a subtype of `StridedMatrix`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> a = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> qr!(a)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
2×2 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.316228  -0.948683
 -0.948683   0.316228
R factor:
2×2 Array{Float64,2}:
 -3.16228  -4.42719
  0.0      -0.632456

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> qr!(a)
ERROR: InexactError: Int64(-3.1622776601683795)
Stacktrace:
[...]
```

LinearAlgebra.LQ — Type.

```
LQ <: Factorization
```

Matrix factorization type of the `LQ` factorization of a matrix `A`. The `LQ` decomposition is the `QR` decomposition of `transpose(A)`. This is the return type of `lq`, the corresponding matrix factorization function.

If `S::LQ` is the factorization object, the lower triangular component can be obtained via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
```

```
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64,Array{Float64,2}} with factors L and Q:
[-8.60233 0.0; 4.41741 -0.697486]
[-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

LinearAlgebra.lq – Function.

```
lq(A) -> S::LQ
```

Compute the LQ decomposition of `A`. The decomposition's lower triangular component can be obtained from the `LQ` object `S` via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

The LQ decomposition is the QR decomposition of `transpose(A)`, and it is useful in order to compute the minimum-norm solution `lq(A) \ b` to an underdetermined system of equations (`A` has more columns than rows, but has full row rank).

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64,Array{Float64,2}} with factors L and Q:
[-8.60233 0.0; 4.41741 -0.697486]
```

```
[-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

LinearAlgebra.lq! – Function.

```
lq!(A) -> LQ
```

Compute the LQ factorization of A, using the input matrix as a workspace. See also lq.

LinearAlgebra.BunchKaufman – Type.

```
BunchKaufman <: Factorization
```

Matrix factorization type of the Bunch-Kaufman factorization of a symmetric or Hermitian matrix A as P'UDU'P or P'LDL'P, depending on whether the upper (the default) or the lower triangle is stored in A. If A is complex symmetric then U' and L' denote the unconjugated transposes, i.e. transpose(U) and transpose(L), respectively. This is the return type of bunchkaufman, the corresponding matrix factorization function.

If S::BunchKaufman is the factorization object, the components can be obtained via S.D, S.U or S.L as appropriate given S.uplo, and S.p.

Iterating the decomposition produces the components S.D, S.U or S.L as appropriate given S.uplo, and S.p.

Examples

```
julia> A = [1 2; 2 3]
2×2 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
```

```
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Array{Int64,1}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ·
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

`LinearAlgebra.bunchkaufman` – Function.

```
bunchkaufman(A, rook::Bool=false; check = true) -> S::BunchKaufman
```

Compute the Bunch-Kaufman [3] factorization of a symmetric or Hermitian matrix `A` as `P'*U*D*U'*P` or `P'*L*D*L'*P`, depending on which triangle is stored in `A`, and return a `BunchKaufman` object. Note that if `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

If `rook` is `true`, rook pivoting is used. If `rook` is false, rook pivoting is not used.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issymmetric`, `ishermitian`, `getindex`.

Examples

```
julia> A = [1 2; 2 3]
2×2 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Array{Int64,1}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
```

[3] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

```
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ·
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

LinearAlgebra.bunchkaufman! – Function.

```
bunchkaufman!(A, rook::Bool=false; check = true) -> BunchKaufman
```

`bunchkaufman!` is the same as bunchkaufman, but saves space by overwriting the input `A`, instead of creating a copy.

LinearAlgebra.Eigen – Type.

```
Eigen <: Factorization
```

Matrix factorization type of the eigenvalue/spectral decomposition of a square matrix `A`. This is the return type of eigen, the corresponding matrix factorization function.

If `F::Eigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
vectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
```

```
 0.0  0.0  1.0

julia> F.values
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

LinearAlgebra.GeneralizedEigen – Type.

```
GeneralizedEigen <: Factorization
```

Matrix factorization type of the generalized eigenvalue/spectral decomposition of `A` and `B`. This is the return type of `eigen`, the corresponding matrix factorization function, when called with two matrix arguments.

If `F::GeneralizedEigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
```

```
 0  1
 1  0

julia> F = eigen(A, B)
GeneralizedEigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Array{Complex{Float64},1}}
values:
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
vectors:
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> F.values
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

LinearAlgebra.eigvals – Function.

```
eigvals(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Return the eigenvalues of A.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The permute, scale, and sortby keywords are the same as for eigen!.

Examples

```
julia> diag_matrix = [1 0; 0 4]
2×2 Array{Int64,2}:
```

```
 1  0
 0  4

julia> eigvals(diag_matrix)
2-element Array{Float64,1}:
 1.0
 4.0
```

For a scalar input, `eigvals` will return a scalar.

Example

```
julia> eigvals(-2)
-2
```

```
eigvals(A, B) -> values
```

Computes the generalized eigenvalues of `A` and `B`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a `UnitRange` `irange` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, 2:2)
1-element Array{Float64,1}:
 0.9999999999999996

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, -1, 2)
1-element Array{Float64,1}:
 1.0000000000000009

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

`LinearAlgebra.eigvals!` – Function.

```
eigvals!(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. The permute, scale, and sortby keywords are the same as for eigen.

> Note
>
> The input matrix A will not contain its eigenvalues after eigvals! is called on it - A is used as a workspace.

Examples

```
julia> A = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> eigvals!(A)
2-element Array{Float64,1}:
 -0.3722813232690143
  5.372281323269014

julia> A
2×2 Array{Float64,2}:
 -0.372281  -1.0
  0.0        5.37228
```

```
eigvals!(A, B; sortby) -> values
```

Same as eigvals, but saves space by overwriting the input A (and B), instead of creating copies.

> Note
>
> The input matrices A and B will not contain their eigenvalues after eigvals! is called. They are used as workspaces.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
```

```
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> eigvals!(A, B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> A
2×2 Array{Float64,2}:
 -0.0  -1.0
  1.0  -0.0

julia> B
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. irange is a range of eigenvalue indices to search for - for instance, the 2nd to 8th eigenvalues.

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. vl is the lower bound of the interval to search for eigenvalues, and vu is the upper bound.

LinearAlgebra.eigmax – Function.

```
eigmax(A; permute::Bool=true, scale::Bool=true)
```

Return the largest eigenvalue of A. The option permute=true permutes the matrix to become closer to upper triangular, and scale=true scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

LinearAlgebra.eigmin – Function.

```
eigmin(A; permute::Bool=true, scale::Bool=true)
```

Return the smallest eigenvalue of `A`. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of `A` are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
```

```
 -1+0im  0+0im


julia> eigmin(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

LinearAlgebra.eigvecs — Function.

```
eigvecs(A::SymTridiagonal[, eigvals]) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

If the optional vector of eigenvalues **`eigvals`** is specified, **`eigvecs`** returns the specific corresponding eigenvectors.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0


julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259


julia> eigvecs(A)
3×3 Array{Float64,2}:
  0.418304  -0.83205      0.364299
 -0.656749  -7.39009e-16  0.754109
  0.627457   0.5547       0.546448


julia> eigvecs(A, [1.])
3×1 Array{Float64,2}:
  0.8320502943378438
  4.263514128092366e-17
 -0.5547001962252291
```

```
eigvecs(A; permute::Bool=true, scale::Bool=true, `sortby`) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.) The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
eigvecs(A, B) -> Matrix
```

Return a matrix `M` whose columns are the generalized eigenvectors of `A` and `B`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im
```

`LinearAlgebra.eigen` — Function.

```
eigen(A; permute::Bool=true, scale::Bool=true, sortby) -> Eigen
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the eigenvalues in an arbitrary order. Some special matrix types (e.g. `Diagonal` or `SymTridiagonal`) may implement their own sorting convention and not accept a `sortby` keyword.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
vectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A, B) -> GeneralizedEigen
```

Computes the generalized eigenvalue decomposition of `A` and `B`, returning a `GeneralizedEigen` factorization object `F` which contains the generalized eigenvalues in `F.values` and the generalized eigenvectors in the columns of the matrix `F.vectors`. (The `k`th generalized eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Any keyword arguments passed to **`eigen`** are passed through to the lower-level `eigen!` function.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> F = eigen(A, B);

julia> F.values
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration
```

```
julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> Eigen
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

The `UnitRange` `irange` specifies indices of the sorted eigenvalues to search for.

> Note
>
> If `irange` is not `1:n`, where `n` is the dimension of `A`, then the returned factorization will be a truncated factorization.

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> Eigen
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The `k`th eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

`vl` is the lower bound of the window of eigenvalues to search for, and `vu` is the upper bound.

> Note
>
> If `[vl, vu]` does not contain all eigenvalues of `A`, then the returned factorization will be a truncated factorization.

`LinearAlgebra.eigen!` – Function.

```
eigen!(A, [B]; permute, scale, sortby)
```

Same as `eigen`, but saves space by overwriting the input `A` (and `B`), instead of creating a copy.

`LinearAlgebra.Hessenberg` – Type.

```
Hessenberg <: Factorization
```

A `Hessenberg` object represents the Hessenberg factorization `QHQ'` of a square matrix, or a shift `Q(H+μI)Q'` thereof, which is produced by the `hessenberg` function.

`LinearAlgebra.hessenberg` – Function.

```
hessenberg(A) -> Hessenberg
```

Compute the Hessenberg decomposition of `A` and return a `Hessenberg` object. If `F` is the factorization object, the unitary matrix can be accessed with `F.Q` (of type `LinearAlgebra.HessenbergQ`) and the Hessenberg matrix with `F.H` (of type `UpperHessenberg`), either of which may be converted to a regular matrix with `Matrix(F.H)` or `Matrix(F.Q)`.

If `A` is `Hermitian` or real-`Symmetric`, then the Hessenberg decomposition produces a real-symmetric tridiagonal matrix and `F.H` is of type `SymTridiagonal`.

Note that the shifted factorization `A+μI = Q (H+μI) Q'` can be constructed efficiently by `F + μ*I` using the `UniformScaling` object `I`, which creates a new `Hessenberg` object with shared storage and a modified shift. The shift of a given `F` is obtained by `F.μ`. This is useful because multiple shifted solves `(F + μ*I) \ b` (for different `μ` and/or `b`) can be performed efficiently once `F` is created.

Iterating the decomposition produces the factors `F.Q, F.H, F.μ`.

Examples

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessenberg(A)
Hessenberg{Float64,UpperHessenberg{Float64,Array{Float64,2}},Array{Float64,2},Array{Float64,1},Bool}
Q factor:
3×3 LinearAlgebra.HessenbergQ{Float64,Array{Float64,2},Array{Float64,1},false}:
 1.0   0.0        0.0
 0.0  -0.707107  -0.707107
 0.0  -0.707107   0.707107
H factor:
3×3 UpperHessenberg{Float64,Array{Float64,2}}:
```

```
 4.0      -11.3137       -1.41421
-5.65685    5.0           2.0
   ·       -8.88178e-16   1.0

julia> F.Q * F.H * F.Q'
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> q, h = F; # destructuring via iteration

julia> q == F.Q && h == F.H
true
```

LinearAlgebra.hessenberg! – Function.

```
hessenberg!(A) -> Hessenberg
```

hessenberg! is the same as hessenberg, but saves space by overwriting the input **A**, instead of creating a copy.

LinearAlgebra.Schur – Type.

```
Schur <: Factorization
```

Matrix factorization type of the Schur factorization of a matrix **A**. This is the return type of schur(_), the corresponding matrix factorization function.

If **F::Schur** is the factorization object, the (quasi) triangular Schur factor can be obtained via either **F.Schur** or **F.T** and the orthogonal/unitary Schur vectors via **F.vectors** or **F.Z** such that **A = F.vectors * F.Schur * F.vectors'**. The eigenvalues of **A** can be obtained with **F.values**.

Iterating the decomposition produces the components **F.T**, **F.Z**, and **F.values**.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
```

```
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

LinearAlgebra.GeneralizedSchur – Type.

```
GeneralizedSchur <: Factorization
```

Matrix factorization type of the generalized Schur factorization of two matrices `A` and `B`. This is the return type of `schur(_, _)`, the corresponding matrix factorization function.

If `F::GeneralizedSchur` is the factorization object, the (quasi) triangular Schur factors can be obtained via `F.S` and `F.T`, the left unitary/orthogonal Schur vectors via `F.left` or `F.Q`, and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

LinearAlgebra.schur – Function.

```
schur(A::StridedMatrix) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the `Schur` object `F` with either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors can be obtained with `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

```
schur(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices `A` and `B`. The (quasi) triangular Schur factors can be obtained from the `Schur` object `F` with `F.S` and `F.T`, the left unitary/orthogonal Schur vectors can be obtained with `F.left` or `F.Q` and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur!` — Function.

```
schur!(A::StridedMatrix) -> F::Schur
```

Same as `schur` but uses the input argument `A` as workspace.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur!(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> A
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
```

```
schur!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as schur but uses the input matrices A and B as workspace.

LinearAlgebra.ordschur — Function.

```
ordschur(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Reorders the Schur factorization F of a matrix A = Z*T*Z' according to the logical array select returning the reordered factorization F object. The selected eigenvalues appear in the leading diagonal of F.Schur and the corresponding leading columns of F.vectors form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via select.

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Reorders the Generalized Schur factorization F of a matrix pair (A, B) = (Q*S*Z', Q*T*Z') according to the logical array select and returns a GeneralizedSchur object F. The selected eigenvalues appear in the leading diagonal of both F.S and F.T, and the left and right orthogonal/unitary Schur vectors are also reordered such that (A, B) = F.Q*(F.S, F.T)*F.Z' still holds and the generalized eigenvalues of A and B can still be obtained with F.α./F.β.

LinearAlgebra.ordschur! — Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as ordschur but overwrites the factorization F.

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Same as ordschur but overwrites the factorization F.

LinearAlgebra.SVD — Type.

```
SVD <: Factorization
```

Matrix factorization type of the singular value decomposition (SVD) of a matrix A. This is the return type of svd(_), the corresponding matrix factorization function.

If F::SVD is the factorization object, U, S, V and Vt can be obtained via F.U, F.S, F.V and F.Vt, such that A = U * Diagonal(S) * Vt. The singular values in S are sorted in descending order.

Iterating the decomposition produces the components U, S, and V.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> F = svd(A)
SVD{Float64,Float64,Array{Float64,2}}
U factor:
4×4 Array{Float64,2}:
 0.0  1.0  0.0   0.0
 1.0  0.0  0.0   0.0
 0.0  0.0  0.0  -1.0
 0.0  0.0  1.0   0.0
singular values:
4-element Array{Float64,1}:
 3.0
 2.23606797749979
 2.0
 0.0
Vt factor:
4×5 Array{Float64,2}:
 -0.0       0.0  1.0  -0.0  0.0
  0.447214  0.0  0.0   0.0  0.894427
 -0.0       1.0  0.0  -0.0  0.0
  0.0       0.0  0.0   1.0  0.0

julia> F.U * Diagonal(F.S) * F.Vt
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> u, s, v = F; # destructuring via iteration

julia> u == F.U && s == F.S && v == F.V
true
```

LinearAlgebra.GeneralizedSVD — Type.

```
GeneralizedSVD <: Factorization
```

Matrix factorization type of the generalized singular value decomposition (SVD) of two matrices `A` and `B`, such that `A = F.U*F.D1*F.R0*F.Q'` and `B = F.V*F.D2*F.R0*F.Q'`. This is the return type of `svd(_, _)`, the corresponding matrix factorization function.

For an M-by-N matrix `A` and P-by-N matrix `B`,

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

`K+L` is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The entries of `F.D1` and `F.D2` are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> F = svd(A, B)
GeneralizedSVD{Float64,Array{Float64,2}}
U factor:
2×2 Array{Float64,2}:
```

```
 1.0  0.0
 0.0  1.0
V factor:
2×2 Array{Float64,2}:
 -0.0  -1.0
  1.0   0.0
Q factor:
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
D1 factor:
2×2 SparseArrays.SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.707107
  [2, 2]  =  0.707107
D2 factor:
2×2 SparseArrays.SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.707107
  [2, 2]  =  0.707107
R0 factor:
2×2 Array{Float64,2}:
 1.41421   0.0
 0.0      -1.41421

julia> F.U*F.D1*F.R0*F.Q'
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0
```

LinearAlgebra.svd – Function.

```
svd(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

Compute the singular value decomposition (SVD) of `A` and return an `SVD` object.

`U`, `S`, `V` and `Vt` can be obtained from the factorization `F` with `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The singular values in `S` are

sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

If `full = false` (default), a "thin" SVD is returned. For a $M \times N$ matrix `A`, in the full factorization `U` is `M \times M` and `V` is `N \times N`, while in the thin factorization `U` is `M \times K` and `V` is `N \times K`, where `K = \min(M,N)` is the number of singular values.

If `alg = DivideAndConquer()` a divide-and-conquer algorithm is used to calculate the SVD. Another (typically slower but more accurate) option is `alg = QRIteration()`.

> Julia 1.3
>
> The `alg` keyword argument requires Julia 1.3 or later.

Examples

```
julia> A = rand(4,3);

julia> F = svd(A); # Store the Factorization Object

julia> A ≈ F.U * Diagonal(F.S) * F.Vt
true

julia> U, S, V = F; # destructuring via iteration

julia> A ≈ U * Diagonal(S) * V'
true

julia> Uonly, = svd(A); # Store U only

julia> Uonly == U
true
```

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` factorization object `F` such that `[A;B] = [F.U * F.D1; F.V * F.D2] * F.R0 * F.Q'`

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,

- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

`K+L` is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The generalized SVD is used in applications such as when one wants to compare how much belongs to `A` vs. how much belongs to `B`, as in human vs yeast genome, or signal vs noise, or between clusters vs within clusters. (See Edelman and Wang for discussion: https://arxiv.org/abs/1901.00485)

It decomposes `[A; B]` into `[UC; VS]H`, where `[UC; VS]` is a natural orthogonal basis for the column space of `[A; B]`, and `H = RQ'` is a natural non-orthogonal basis for the rowspace of `[A;B]`, where the top rows are most closely attributed to the `A` matrix, and the bottom to the `B` matrix. The multi-cosine/sine matrices `C` and `S` provide a multi-measure of how much `A` vs how much `B`, and `U` and `V` provide directions in which these are measured.

Examples

```
julia> A = randn(3,2); B=randn(4,2);

julia> F = svd(A, B);

julia> U,V,Q,C,S,R = F;

julia> H = R*Q';

julia> [A; B] ≈ [U*C; V*S]*H
true

julia> [A; B] ≈ [F.U*F.D1; F.V*F.D2]*F.R0*F.Q'
true

julia> Uonly, = svd(A,B);

julia> U == Uonly
true
```

`LinearAlgebra.svd!` — Function.

```
svd!(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

`svd!` is the same as svd, but saves space by overwriting the input `A`, instead of creating a copy. See documentation of svd for details. “‘

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as svd, but modifies the arguments `A` and `B` in-place, instead of making copies. See documentation of svd for details. “‘

`LinearAlgebra.svdvals` — Function.

```
svdvals(A)
```

Return the singular values of `A` in descending order.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> svdvals(A)
4-element Array{Float64,1}:
 3.0
 2.23606797749979
 2.0
 0.0
```

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of `A` and `B`. See also svd.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
```

```
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> svdvals(A, B)
2-element Array{Float64,1}:
 1.0
 1.0
```

LinearAlgebra.svdvals! – Function.

```
svdvals!(A)
```

Return the singular values of `A`, saving space by overwriting the input. See also svdvals and svd. “‘

```
svdvals!(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of `A` and `B`, saving space by overwriting `A` and `B`. See also svd and svdvals. “‘

LinearAlgebra.Givens – Type.

```
LinearAlgebra.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields `c` and `s` represent the cosine and sine of the rotation angle, respectively. The `Givens` type supports left multiplication `G*A` and conjugated transpose right multiplication `A*G'`. The type doesn't have a `size` and can therefore be multiplied with matrices of arbitrary size as long as `i2<=size(A,2)` for `G*A` or `i2<=size(A,1)` for `A*G'`.

See also: givens

LinearAlgebra.givens – Function.

```
givens(f::T, g::T, i1::Integer, i2::Integer) where {T} -> (G::Givens, r::T)
```

Computes the Givens rotation `G` and scalar `r` such that for any vector `x` where

```
x[i1] = f
x[i2] = g
```

the result of the multiplication

```
y = G*x
```

has the property that

```
y[i1] = r
y[i2] = 0
```

See also: LinearAlgebra.Givens

```
givens(A::AbstractArray, i1::Integer, i2::Integer, j::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*A
```

has the property that

```
B[i1,j] = r
B[i2,j] = 0
```

See also: LinearAlgebra.Givens

```
givens(x::AbstractVector, i1::Integer, i2::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*x
```

has the property that

```
B[i1] = r
B[i2] = 0
```

See also: LinearAlgebra.Givens

LinearAlgebra.triu – Function.

```
triu(M)
```

Upper triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0
 0.0  0.0  0.0  1.0
```

```
triu(M, k::Integer)
```

Returns the upper triangle of `M` starting from the `k`th superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a,3)
4×4 Array{Float64,2}:
 0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> triu(a,-3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

LinearAlgebra.triu! – Function.

```
triu!(M)
```

Upper triangle of a matrix, overwriting `M` in the process. See also `triu`.

```
triu!(M, k::Integer)
```

Return the upper triangle of `M` starting from the `k`th superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> triu!(M, 1)
5×5 Array{Int64,2}:
 0  2  3  4  5
 0  0  3  4  5
 0  0  0  4  5
 0  0  0  0  5
 0  0  0  0  0
```

LinearAlgebra.tril – Function.

```
tril(M)
```

Lower triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
julia> tril(a)
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 1.0  1.0  0.0  0.0
 1.0  1.0  1.0  0.0
 1.0  1.0  1.0  1.0
```

```
tril(M, k::Integer)
```

Returns the lower triangle of `M` starting from the `k`th superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,-3)
4×4 Array{Float64,2}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

`LinearAlgebra.tril!` — Function.

```
tril!(M)
```

Lower triangle of a matrix, overwriting `M` in the process. See also `tril`.

```
tril!(M, k::Integer)
```

Return the lower triangle of `M` starting from the `k`th superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> tril!(M, 2)
5×5 Array{Int64,2}:
 1  2  3  0  0
 1  2  3  4  0
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

`LinearAlgebra.diagind` — Function.

```
diagind(M, k::Integer=0)
```

An `AbstractRange` giving the indices of the `k`th diagonal of the matrix `M`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diagind(A,-1)
2:4:6
```

`LinearAlgebra.diag` — Function.

```
diag(M, k::Integer=0)
```

The kth diagonal of a matrix, as a vector.

See also: diagm

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
2-element Array{Int64,1}:
 2
 6
```

LinearAlgebra.diagm – Function.

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
diagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a matrix from Pairs of diagonals and vectors. Vector kv.second will be placed on the kv.first diagonal. By default the matrix is square and its size is inferred from kv, but a non-square size m×n (padded with zeros as needed) can be specified by passing m,n as the first arguments.

diagm constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see Diagonal, Bidiagonal Tridiagonal and SymTridiagonal.

Examples

```
julia> diagm(1 => [1,2,3])
4×4 Array{Int64,2}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], -1 => [4,5])
4×4 Array{Int64,2}:
 0  1  0  0
 4  0  2  0
```

```
 0  5  0  3
 0  0  0  0
```

```
diagm(v::AbstractVector)
diagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a matrix with elements of the vector as diagonal elements. By default (if `size=nothing`), the matrix is square and its size is given by `length(v)`, but a non-square size `m×n` can be specified by passing `m,n` as the first arguments.

Examples

```
julia> diagm([1,2,3])
3×3 Array{Int64,2}:
 1  0  0
 0  2  0
 0  0  3
```

`LinearAlgebra.rank` — Function.

```
rank(A::AbstractMatrix; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
rank(A::AbstractMatrix, rtol::Real)
```

Compute the rank of a matrix by counting how many singular values of `A` have magnitude greater than `max(atol, rtol*σ₁)` where `σ₁` is `A`'s largest singular value. `atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `A`, and `ϵ` is the `eps` of the element type of `A`.

> Julia 1.1
>
> The `atol` and `rtol` keyword arguments requires at least Julia 1.1. In Julia 1.0 `rtol` is available as a positional argument, but this will be deprecated in Julia 2.0.

Examples

```
julia> rank(Matrix(I, 3, 3))
3


julia> rank(diagm(0 => [1, 0, 2]))
2
```

```
julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.1)
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.00001)
3

julia> rank(diagm(0 => [1, 0.001, 2]), atol=1.5)
1
```

LinearAlgebra.norm – Function.

```
norm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the `p`-norm (defaulting to `p=2`) as if `A` were a vector of the corresponding length.

The `p`-norm is defined as

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A, $|a_i|$ the norm of a_i, and n the length of A. Since the `p`-norm is computed using the norms of the entries of `A`, the `p`-norm of a vector of vectors is not compatible with the interpretation of it as a block vector in general if `p != 2`.

`p` can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs.(A)`, whereas `norm(A, -Inf)` returns the smallest. If `A` is a matrix and `p=2`, then this is equivalent to the Frobenius norm.

The second argument `p` is not necessarily a part of the interface for `norm`, i.e. a custom type may only implement `norm(A)` without second argument.

Use opnorm to compute the operator norm of a matrix.

Examples

```
julia> v = [3, -2, 6]
3-element Array{Int64,1}:
  3
 -2
  6
```

```
julia> norm(v)
7.0

julia> norm(v, 1)
11.0

julia> norm(v, Inf)
6.0

julia> norm([1 2 3; 4 5 6; 7 8 9])
16.881943016134134

julia> norm([1 2 3 4 5 6 7 8 9])
16.881943016134134

julia> norm(1:9)
16.881943016134134

julia> norm(hcat(v,v), 1) == norm(vcat(v,v), 1) != norm([v,v], 1)
true

julia> norm(hcat(v,v), 2) == norm(vcat(v,v), 2) == norm([v,v], 2)
true

julia> norm(hcat(v,v), Inf) == norm(vcat(v,v), Inf) != norm([v,v], Inf)
true
```

```
norm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p \right)^{1/p}$.

Examples

```
julia> norm(2, 1)
2.0

julia> norm(-2, 1)
2.0

julia> norm(2, 2)
```

```
2.0

julia> norm(-2, 2)
2.0

julia> norm(2, Inf)
2.0

julia> norm(-2, Inf)
2.0
```

LinearAlgebra.opnorm – Function.

```
opnorm(A::AbstractMatrix, p::Real=2)
```

Compute the operator norm (or matrix norm) induced by the vector `p`-norm, where valid values of `p` are `1`, `2`, or `Inf`. (Note that for sparse matrices, `p=2` is currently not implemented.) Use `norm` to compute the Frobenius norm.

When `p=1`, the operator norm is the maximum absolute column sum of `A`:

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When `p=2`, the operator norm is the spectral norm, equal to the largest singular value of `A`.

When `p=Inf`, the operator norm is the maximum absolute row sum of `A`:

$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|$$

Examples

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Array{Int64,2}:
 1  -2  -3
 2   3  -1

julia> opnorm(A, Inf)
6.0
```

```
julia> opnorm(A, 1)
5.0
```

```
opnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to `norm`.

```
opnorm(A::Adjoint{<:Any,<:AbstracVector}, q::Real=2)
opnorm(A::Transpose{<:Any,<:AbstracVector}, q::Real=2)
```

For Adjoint/Transpose-wrapped vectors, return the operator q-norm of `A`, which is equivalent to the `p`-norm with value `p = q/(q-1)`. They coincide at `p = q = 2`. Use `norm` to compute the `p` norm of `A` as a vector.

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the dot product, and the result is consistent with the operator `p`-norm of a `1 × n` matrix.

Examples

```
julia> v = [1; im];

julia> vc = v';

julia> opnorm(vc, 1)
1.0

julia> norm(vc, 1)
2.0

julia> norm(v, 1)
2.0

julia> opnorm(vc, 2)
1.4142135623730951

julia> norm(vc, 2)
1.4142135623730951

julia> norm(v, 2)
1.4142135623730951

julia> opnorm(vc, Inf)
```

```
2.0

julia> norm(vc, Inf)
1.0

julia> norm(v, Inf)
1.0
```

LinearAlgebra.normalize! – Function.

```
normalize!(a::AbstractArray, p::Real=2)
```

Normalize the array `a` in-place so that its `p`-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize` and `norm`.

LinearAlgebra.normalize – Function.

```
normalize(a::AbstractArray, p::Real=2)
```

Normalize the array `a` so that its `p`-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize!` and `norm`.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Array{Float64,1}:
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Array{Float64,1}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714

julia> norm(c, 1)
```

```
1.0

julia> a = [1 2 4 ; 1 2 4]
2×3 Array{Int64,2}:
 1  2  4
 1  2  4

julia> norm(a)
6.48074069840786

julia> normalize(a)
2×3 Array{Float64,2}:
 0.154303  0.308607  0.617213
 0.154303  0.308607  0.617213
```

LinearAlgebra.cond – Function.

```
cond(M, p::Real=2)
```

Condition number of the matrix `M`, computed using the operator `p`-norm. Valid values for `p` are `1`, `2` (default), or `Inf`.

LinearAlgebra.condskeel – Function.

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M,p) = \left\| |M| \left|M^{-1}\right| \right\|_p$$
$$\kappa_S(M,x,p) = \frac{\left\| |M| \left|M^{-1}\right| |x| \right\|_p}{\|x\|_p}$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator `p`-norm. $|M|$ denotes the matrix of (entry wise) absolute values of M; $|M|_{ij} = |M_{ij}|$. Valid values for `p` are `1`, `2` and `Inf` (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

LinearAlgebra.tr – Function.

```
tr(M)
```

Matrix trace. Sums the diagonal elements of `M`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> tr(A)
5
```

`LinearAlgebra.det` – Function.

```
det(M)
```

Matrix determinant.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> det(M)
2.0
```

`LinearAlgebra.logdet` – Function.

```
logdet(M)
```

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> logdet(M)
```

```
0.6931471805599453

julia> logdet(Matrix(I, 3, 3))
0.0
```

LinearAlgebra.logabsdet – Function.

```
logabsdet(M)
```

Log of absolute value of matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

Examples

```
julia> A = [-1. 0.; 0. 1.]
2×2 Array{Float64,2}:
 -1.0  0.0
  0.0  1.0

julia> det(A)
-1.0

julia> logabsdet(A)
(0.0, -1.0)

julia> B = [2. 0.; 0. 1.]
2×2 Array{Float64,2}:
 2.0  0.0
 0.0  1.0

julia> det(B)
2.0

julia> logabsdet(B)
(0.6931471805599453, 1.0)
```

Base.inv – Method.

```
inv(M)
```

Matrix inverse. Computes matrix `N` such that `M * N = I`, where `I` is the identity matrix. Computed by solving the left-division `N = M \ I`.

Examples

```
julia> M = [2 5; 1 3]
2×2 Array{Int64,2}:
 2  5
 1  3

julia> N = inv(M)
2×2 Array{Float64,2}:
  3.0  -5.0
 -1.0   2.0

julia> M*N == N*M == Matrix(I, 2, 2)
true
```

LinearAlgebra.pinv – Function.

```
pinv(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
pinv(M, rtol::Real) = pinv(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes the Moore-Penrose pseudoinverse.

For matrices `M` with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values greater than `max(atol, rtol*σ₁)` where `σ₁` is the largest singular value of `M`.

The optimal choice of absolute (`atol`) and relative tolerance (`rtol`) varies both with the value of `M` and the intended application of the pseudoinverse. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the eps of the element type of `M`.

For inverting dense ill-conditioned matrices in a least-squares sense, `rtol = sqrt(eps(real(float(one(eltype(M))))))` is recommended.

For more information, see [4], [5], [6], [7].

Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Array{Float64,2}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
```

```
2×2 Array{Float64,2}:
  1.47287   -1.00775
 -0.930233   1.16279

julia> M * N
2×2 Array{Float64,2}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

`LinearAlgebra.nullspace` — Function.

```
nullspace(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
nullspace(M, rtol::Real) = nullspace(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes a basis for the nullspace of `M` by including the singular vectors of `M` whose singular values have magnitudes greater than `max(atol, rtol*σ₁)`, where `σ₁` is `M`'s largest singular value.

By default, the relative tolerance `rtol` is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the `eps` of the element type of `M`.

Examples

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

[4] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[5] ☒ke Bj☒rck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[6] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[7] Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585

```
julia> nullspace(M, rtol=3)
3×3 Array{Float64,2}:
 0.0  1.0  0.0
 1.0  0.0  0.0
 0.0  0.0  1.0

julia> nullspace(M, atol=0.95)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

`Base.kron` – Function.

```
kron(A, B)
```

Kronecker tensor product of two vectors or two matrices.

For real vectors `v` and `w`, the Kronecker product is related to the outer product by `kron(v,w) == vec(w * transpose(v))` or `w * transpose(v) == reshape(kron(v,w), (length(w), length(v)))`. Note how the ordering of `v` and `w` differs on the left and right of these expressions (due to column-major storage). For complex vectors, the outer product `w * v'` also differs by conjugation of `v`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [im 1; 1 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Array{Complex{Int64},2}:
 0+1im  1+0im  0+2im  2+0im
 1+0im  0-1im  2+0im  0-2im
 0+3im  3+0im  0+4im  4+0im
 3+0im  0-3im  4+0im  0-4im
```

```
julia> v = [1, 2]; w = [3, 4, 5];

julia> w*transpose(v)
3×2 Array{Int64,2}:
 3   6
 4   8
 5  10

julia> reshape(kron(v,w), (length(w), length(v)))
3×2 Array{Int64,2}:
 3   6
 4   8
 5  10
```

`Base.exp` — Method.

```
exp(A::AbstractMatrix)
```

Compute the matrix exponential of `A`, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian `A`, an eigendecomposition (`eigen`) is used, otherwise the scaling and squaring algorithm (see [8]) is chosen.

Examples

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> exp(A)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828
```

[8] Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539

Base.:^ – Method.

```
^(A::AbstractMatrix, p::Number)
```

Matrix power, equivalent to $\exp(p \log(A))$

Examples

```
julia> [1 2; 0 3]^3
2×2 Array{Int64,2}:
 1  26
 0  27
```

Base.:^ – Method.

```
^(b::Number, A::AbstractMatrix)
```

Matrix exponential, equivalent to $\exp(\log(b)A)$.

> Julia 1.1
>
> Support for raising `Irrational` numbers (like) to a matrix was added in Julia 1.1.

Examples

```
julia> 2^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.0  6.0
 0.0  8.0

julia>  ^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.71828  17.3673
 0.0      20.0855
```

Base.log – Method.

```
log(A{T}::StridedMatrix{T})
```

If `A` has no negative real eigenvalue, compute the principal matrix logarithm of `A`, i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If `A` has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used, if `A` is triangular an improved version of the inverse scaling and squaring method is employed (see [9] and [10]). For general matrices, the complex Schur form (`schur`) is computed and the triangular algorithm is used on the triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828

julia> log(A)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

`Base.sqrt` – Method.

```
sqrt(A::AbstractMatrix)
```

If `A` has no negative real eigenvalues, compute the principal matrix square root of `A`, that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If `A` is real-symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the square root. For such matrices, eigenvalues λ that appear to be slightly negative due to roundoff errors are treated as if they were zero More precisely, matrices with all eigenvalues `≥ -rtol*(max |λ|)` are treated as semidefinite (yielding a Hermitian square root), with negative eigenvalues taken to be zero. `rtol` is a keyword argument to `sqrt` (in the Hermitian/real-symmetric case only) that defaults to machine precision scaled by `size(A,1)`.

Otherwise, the square root is determined by means of the Bj□rck-Hammarling method [11], which computes the complex Schur form (`schur`) and then the complex square root of the triangular factor.

Examples

```
julia> A = [4 0; 0 4]
2×2 Array{Int64,2}:
```

[9] Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553

[10] Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fr□chet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

[11] □ke Bj□rck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X

```
 4  0
 0  4

julia> sqrt(A)
2×2 Array{Float64,2}:
 2.0  0.0
 0.0  2.0
```

Base.cos – Method.

```
cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the cosine. Otherwise, the cosine is determined by calling exp.

Examples

```
julia> cos(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Base.sin – Method.

```
sin(A::AbstractMatrix)
```

Compute the matrix sine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the sine. Otherwise, the sine is determined by calling exp.

Examples

```
julia> sin(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649
```

Base.Math.sincos – Method.

```
sincos(A::AbstractMatrix)
```

Compute the matrix sine and cosine of a square matrix `A`.

Examples

```
julia> S, C = sincos(fill(1.0, (2,2)));

julia> S
2×2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649

julia> C
2×2 Array{Float64,2}:
  0.291927  -0.708073
 -0.708073   0.291927
```

`Base.tan` – Method.

```
tan(A::AbstractMatrix)
```

Compute the matrix tangent of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the tangent. Otherwise, the tangent is determined by calling `exp`.

Examples

```
julia> tan(fill(1.0, (2,2)))
2×2 Array{Float64,2}:
 -1.09252  -1.09252
 -1.09252  -1.09252
```

`Base.Math.sec` – Method.

```
sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix `A`.

`Base.Math.csc` – Method.

```
csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix `A`.

Base.Math.cot – Method.

```
cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix A.

Base.cosh – Method.

```
cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix A.

Base.sinh – Method.

```
sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix A.

Base.tanh – Method.

```
tanh(A::AbstractMatrix)
```

Compute the matrix hyperbolic tangent of a square matrix A.

Base.Math.sech – Method.

```
sech(A::AbstractMatrix)
```

Compute the matrix hyperbolic secant of square matrix A.

Base.Math.csch – Method.

```
csch(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosecant of square matrix A.

Base.Math.coth – Method.

```
coth(A::AbstractMatrix)
```

Compute the matrix hyperbolic cotangent of square matrix A.

Base.acos – Method.

```
acos(A::AbstractMatrix)
```

Compute the inverse matrix cosine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse cosine. Otherwise, the inverse cosine is determined by using `log` and `sqrt`. For the theory and logarithmic formulas used to compute this function, see [12].

Examples

```
julia> acos(cos([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5-8.32667e-17im  0.1+0.0im
 -0.2+2.63678e-16im  0.3-3.46945e-16im
```

`Base.asin` — Method.

```
asin(A::AbstractMatrix)
```

Compute the inverse matrix sine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse sine. Otherwise, the inverse sine is determined by using `log` and `sqrt`. For the theory and logarithmic formulas used to compute this function, see [13].

Examples

```
julia> asin(sin([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5-4.16334e-17im  0.1-5.55112e-17im
 -0.2+9.71445e-17im  0.3-1.249e-16im
```

`Base.atan` — Method.

```
atan(A::AbstractMatrix)
```

[12] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[13] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

Compute the inverse matrix tangent of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the inverse tangent. Otherwise, the inverse tangent is determined by using `log`. For the theory and logarithmic formulas used to compute this function, see [14].

Examples

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  0.5+1.38778e-17im  0.1-2.77556e-17im
 -0.2+6.93889e-17im  0.3-4.16334e-17im
```

`Base.Math.asec` – Method.

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of `A`.

`Base.Math.acsc` – Method.

```
acsc(A::AbstractMatrix)
```

Compute the inverse matrix cosecant of `A`.

`Base.Math.acot` – Method.

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of `A`.

`Base.acosh` – Method.

```
acosh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix cosine of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [15].

[14] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[15] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

Base.asinh – Method.

```
asinh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix sine of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [16].

Base.atanh – Method.

```
atanh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix tangent of a square matrix `A`. For the theory and logarithmic formulas used to compute this function, see [17].

Base.Math.asech – Method.

```
asech(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic secant of `A`.

Base.Math.acsch – Method.

```
acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of `A`.

Base.Math.acoth – Method.

```
acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of `A`.

LinearAlgebra.lyap – Function.

```
lyap(A, C)
```

Computes the solution `X` to the continuous Lyapunov equation `AX + XA' + C = 0`, where no eigenvalue of `A` has a zero real part and no two eigenvalues are negative complex conjugates of each other.

Examples

[16] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[17] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

```
julia> A = [3. 4.; 5. 6]
2×2 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> X = lyap(A, B)
2×2 Array{Float64,2}:
  0.5  -0.5
 -0.5   0.25

julia> A*X + X*A' + B
2×2 Array{Float64,2}:
 0.0          6.66134e-16
 6.66134e-16  8.88178e-16
```

`LinearAlgebra.sylvester` – Function.

```
sylvester(A, B, C)
```

Computes the solution `X` to the Sylvester equation `AX + XB + C = 0`, where `A`, `B` and `C` have compatible dimensions and `A` and `-B` have no eigenvalues with equal real part.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> C = [1. 2.; -2. 1]
2×2 Array{Float64,2}:
```

```
 1.0  2.0
-2.0  1.0

julia> X = sylvester(A, B, C)
2×2 Array{Float64,2}:
 -4.46667   1.93333
  3.73333  -1.8

julia> A*X + X*B + C
2×2 Array{Float64,2}:
  2.66454e-15  1.77636e-15
 -3.77476e-15  4.44089e-16
```

LinearAlgebra.issuccess – Function.

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

```
julia> F = cholesky([1 0; 0 1]);

julia> LinearAlgebra.issuccess(F)
true

julia> F = lu([1 0; 0 0]; check = false);

julia> LinearAlgebra.issuccess(F)
false
```

LinearAlgebra.issymmetric – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1
```

```
julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> issymmetric(b)
false
```

LinearAlgebra.isposdef – Function.

```
isposdef(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of `A`. See also `isposdef!`

Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1   2
 2  50

julia> isposdef(A)
true
```

LinearAlgebra.isposdef! – Function.

```
isposdef!(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of `A`, overwriting `A` in the process. See also `isposdef`.

Examples

```
julia> A = [1. 2.; 2. 50.];

julia> isposdef!(A)
```

```
true


julia> A
2×2 Array{Float64,2}:
 1.0  2.0
 2.0  6.78233
```

LinearAlgebra.istril – Function.

```
istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is lower triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istril(a)
false

julia> istril(a, 1)
true

julia> b = [1 0; -im -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+0im
 0-1im  -1+0im

julia> istril(b)
true

julia> istril(b, -1)
false
```

LinearAlgebra.istriu – Function.

```
istriu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether `A` is upper triangular starting from the `k`th superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istriu(a)
false

julia> istriu(a, -1)
true

julia> b = [1 im; 0 -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+1im
 0+0im  -1+0im

julia> istriu(b)
true

julia> istriu(b, 1)
false
```

`LinearAlgebra.isdiag` – Function.

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> isdiag(a)
false
```

```
julia> b = [im 0; 0 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true
```

LinearAlgebra.ishermitian – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

Base.transpose – Function.

```
transpose(A)
```

Lazy transpose. Mutating the returned object should appropriately mutate `A`. Often, but not always, yields `Transpose(A)`, where `Transpose` is a lazy transpose wrapper. Note that this operation is recursive.

This operation is intended for linear algebra usage - for general data manipulation see permutedims, which is non-recursive.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

`LinearAlgebra.transpose!` – Function.

```
transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> transpose!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3+2im  8+7im
 9+2im  4+6im

julia> A
2×2 Array{Complex{Int64},2}:
```

```
 3+2im  9+2im
 8+7im  4+6im
```

LinearAlgebra.Transpose – Type.

```
Transpose
```

Lazy wrapper type for a transpose view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`, but also some `Factorization`, for instance. Usually, the `Transpose` constructor should not be called directly, use `transpose` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

Base.adjoint – Function.

```
A'
adjoint(A)
```

Lazy adjoint (conjugate transposition). Note that `adjoint` is applied recursively to elements.

For number types, `adjoint` returns the complex conjugate, and therefore it is equivalent to the identity function for real numbers.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
```

```
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im

julia> x = [3, 4im]
2-element Array{Complex{Int64},1}:
 3 + 0im
 0 + 4im

julia> x'x
25 + 0im
```

LinearAlgebra.adjoint! – Function.

```
adjoint!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> adjoint!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3-2im  8-7im
 9-2im  4-6im
```

```
julia> A
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im
```

LinearAlgebra.Adjoint – Type.

```
Adjoint
```

Lazy wrapper type for an adjoint view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`, but also some `Factorization`, for instance. Usually, the `Adjoint` constructor should not be called directly, use `adjoint` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im
```

Base.copy – Method.

```
copy(A::Transpose)
copy(A::Adjoint)
```

Eagerly evaluate the lazy matrix transpose/adjoint. Note that the transposition is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [1 2im; -3im 4]
2×2 Array{Complex{Int64},2}:
```

```
 1+0im  0+2im
 0-3im  4+0im

julia> T = transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Array{Complex{Int64},2}:
 1+0im  0-3im
 0+2im  4+0im
```

LinearAlgebra.stride1 – Function.

```
stride1(A) -> Int
```

Return the distance between successive array elements in dimension 1 in units of element size.

Examples

```
julia> A = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> LinearAlgebra.stride1(A)
1

julia> B = view(A, 2:2:4)
2-element view(::Array{Int64,1}, 2:2:4) with eltype Int64:
 2
 4

julia> LinearAlgebra.stride1(B)
2
```

LinearAlgebra.checksquare – Function.

```
LinearAlgebra.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

```
julia> A = fill(1, (4,4)); B = fill(1, (5,5));


julia> LinearAlgebra.checksquare(A, B)
2-element Array{Int64,1}:
 4
 5
```

LinearAlgebra.peakflops — Function.

```
LinearAlgebra.peakflops(n::Integer=2000; parallel::Bool=false)
```

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies a matrix of size `n x n`, where `n = 2000`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

> Julia 1.1
>
> This function requires at least Julia 1.1. In Julia 1.0 it is available from the standard library `InteractiveUtils`.

77.4 Low-level matrix operations

In many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with `!` below (e.g. `mul!`) according to the usual Julia convention.

LinearAlgebra.mul! — Function.

```
mul!(Y, A, B) -> Y
```

Calculates the matrix-matrix or matrix-vector product AB and stores the result in `Y`, overwriting the existing value of `Y`. Note that `Y` must not be aliased with either `A` or `B`.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; Y = similar(B); mul!(Y, A, B);

julia> Y
2×2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

Implementation

For custom matrix and vector types, it is recommended to implement 5-argument `mul!` rather than implementing 3-argument `mul!` directly if possible.

```
mul!(C, A, B, α, β) -> C
```

Combined inplace matrix-matrix or matrix-vector multiply-add $AB+C$. The result is stored in `C` by overwriting it. Note that `C` must not be aliased with either `A` or `B`.

> Julia 1.3
>
> Five-argument `mul!` requires at least Julia 1.3.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; C=[1.0 2.0; 3.0 4.0];

julia> mul!(C, A, B, 100.0, 10.0) === C
true

julia> C
2×2 Array{Float64,2}:
 310.0  320.0
 730.0  740.0
```

`LinearAlgebra.lmul!` — Function.

```
lmul!(a::Number, B::AbstractArray)
```

Scale an array `B` by a scalar `a` overwriting `B` in-place. Use `rmul!` to multiply scalar from right. The scaling operation respects the semantics of the multiplication `*` between `a` and an element of `B`. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

> Julia 1.1
>
> Prior to Julia 1.1, `NaN` and `±Inf` entries in `B` were treated inconsistently.

Examples

```
julia> B = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> lmul!(2, B)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> lmul!(0.0, [Inf])
1-element Array{Float64,1}:
 NaN
```

```
lmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting `B`, and return the result. Here, `A` must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> B = [0 1; 1 0];

julia> A = LinearAlgebra.UpperTriangular([1 2; 0 3]);

julia> LinearAlgebra.lmul!(A, B);

julia> B
2×2 Array{Int64,2}:
 2  1
 3  0

julia> B = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> lmul!(F.Q, B)
2×2 Array{Float64,2}:
```

```
 3.0  4.0
 1.0  2.0
```

LinearAlgebra.rmul! – Function.

```
rmul!(A::AbstractArray, b::Number)
```

Scale an array A by a scalar b overwriting A in-place. Use lmul! to multiply scalar from left. The scaling operation respects the semantics of the multiplication * between an element of A and b. In particular, this also applies to multiplication involving non-finite numbers such as NaN and ±Inf.

Julia 1.1

Prior to Julia 1.1, NaN and ±Inf entries in A were treated inconsistently.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rmul!(A, 2)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> rmul!([NaN], 0.0)
1-element Array{Float64,1}:
 NaN
```

```
rmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting A, and return the result. Here, B must be of special matrix type, like, e.g., Diagonal, UpperTriangular or LowerTriangular, or of some orthogonal type, see QR.

Examples

```
julia> A = [0 1; 1 0];

julia> B = LinearAlgebra.UpperTriangular([1 2; 0 3]);
```

```
julia> LinearAlgebra.rmul!(A, B);

julia> A
2×2 Array{Int64,2}:
 0  3
 1  2

julia> A = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> rmul!(A, F.Q)
2×2 Array{Float64,2}:
 2.0  1.0
 4.0  3.0
```

LinearAlgebra.ldiv! – Function.

```
ldiv!(Y, A, B) -> Y
```

Compute `A \ B` in-place and store the result in `Y`, returning the result.

The argument `A` should not be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by factorize or cholesky). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., lu!), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = zero(X);

julia> ldiv!(Y, qr(A), X);

julia> Y
3-element Array{Float64,1}:
  0.7128099173553719
```

```
 -0.051652892561983674
  0.10020661157024757

julia> A\X
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
ldiv!(A, B)
```

Compute `A \ B` in-place and overwriting `B` to store the result.

The argument `A` should not be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = copy(X);

julia> ldiv!(qr(A), X);

julia> X
3-element Array{Float64,1}:
  0.7128099173553719
 -0.051652892561983674
  0.10020661157024757

julia> A\Y
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
ldiv!(a::Number, B::AbstractArray)
```

Divide each entry in an array `B` by a scalar `a` overwriting `B` in-place. Use `rdiv!` to divide scalar from right.

Examples

```
julia> B = [1.0 2.0; 3.0 4.0]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> ldiv!(2.0, B)
2×2 Array{Float64,2}:
 0.5  1.0
 1.5  2.0
```

`LinearAlgebra.rdiv!` — Function.

```
rdiv!(A, B)
```

Compute `A / B` in-place and overwriting `A` to store the result.

The argument `B` should not be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `rdiv!` usually also require fine-grained control over the factorization of `B`.

```
rdiv!(A::AbstractArray, b::Number)
```

Divide each entry in an array `A` by a scalar `b` overwriting `A` in-place. Use `ldiv!` to divide scalar from left.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> rdiv!(A, 2.0)
2×2 Array{Float64,2}:
 0.5  1.0
 1.5  2.0
```

77.5 BLAS Functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the LAPACK library, which in turn is built on top of basic linear-algebra building-blocks known as the BLAS. There are highly optimized implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

`LinearAlgebra.BLAS` provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in `'!'`. Usually, a BLAS function has four methods defined, for `Float64`, `Float32`, `ComplexF64`, and `ComplexF32` arrays.

BLAS Character Arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (`trans`), which triangle of a matrix to reference (`uplo` or `ul`), whether the diagonal of a triangular matrix can be assumed to be all ones (`dA`) or which side of a matrix multiplication the input argument belongs on (`side`). The possibilities are:

Multplication Order

side	Meaning
'L'	The argument goes on the left side of a matrix-matrix operation.
'R'	The argument goes on the right side of a matrix-matrix operation.

Triangle Referencing

uplo/ul	Meaning
'U'	Only the upper triangle of the matrix will be used.
'L'	Only the lower triangle of the matrix will be used.

Transposition Operation

trans/tX	Meaning
'N'	The input matrix X is not transposed or conjugated.
'T'	The input matrix X will be transposed.
'C'	The input matrix X will be conjugated and transposed.

Unit Diagonal

`LinearAlgebra.BLAS` – Module.

diag/dX	Meaning
'N'	The diagonal values of the matrix X will be read.
'U'	The diagonal of the matrix X is assumed to be all ones.

Interface to BLAS subroutines.

LinearAlgebra.BLAS.dot – Function.

```
dot(n, X, incx, Y, incy)
```

Dot product of two vectors consisting of n elements of array X with stride incx and n elements of array Y with stride incy.

Examples

```
julia> BLAS.dot(10, fill(1.0, 10), 1, fill(1.0, 20), 2)
10.0
```

LinearAlgebra.BLAS.dotu – Function.

```
dotu(n, X, incx, Y, incy)
```

Dot function for two complex vectors consisting of n elements of array X with stride incx and n elements of array Y with stride incy.

Examples

```
julia> BLAS.dotu(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
-10.0 + 10.0im
```

LinearAlgebra.BLAS.dotc – Function.

```
dotc(n, X, incx, U, incy)
```

Dot function for two complex vectors, consisting of n elements of array X with stride incx and n elements of array U with stride incy, conjugating the first vector.

Examples

```
julia> BLAS.dotc(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
10.0 - 10.0im
```

LinearAlgebra.BLAS.blascopy! – Function.

```
blascopy!(n, X, incx, Y, incy)
```

Copy `n` elements of array `X` with stride `incx` to array `Y` with stride `incy`. Returns `Y`.

LinearAlgebra.BLAS.nrm2 – Function.

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

Examples

```
julia> BLAS.nrm2(4, fill(1.0, 8), 2)
2.0

julia> BLAS.nrm2(1, fill(1.0, 8), 2)
1.0
```

LinearAlgebra.BLAS.asum – Function.

```
asum(n, X, incx)
```

Sum of the absolute values of the first `n` elements of array `X` with stride `incx`.

Examples

```
julia> BLAS.asum(5, fill(1.0im, 10), 2)
5.0

julia> BLAS.asum(2, fill(1.0im, 10), 5)
2.0
```

LinearAlgebra.axpy! – Function.

```
axpy!(a, X, Y)
```

Overwrite `Y` with `X*a + Y`, where `a` is a scalar. Return `Y`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> BLAS.axpy!(2, x, y)
3-element Array{Int64,1}:
  6
  9
 12
```

LinearAlgebra.axpby! – Function.

```
axpby!(a, X, b, Y)
```

Overwrite `Y` with `X*a + Y*b`, where `a` and `b` are scalars. Return `Y`.

Examples

```
julia> x = [1., 2, 3];

julia> y = [4., 5, 6];

julia> BLAS.axpby!(2., x, 3., y)
3-element Array{Float64,1}:
 14.0
 19.0
 24.0
```

LinearAlgebra.BLAS.scal! – Function.

```
scal!(n, a, X, incx)
```

Overwrite `X` with `a*X` for the first `n` elements of array `X` with stride `incx`. Returns `X`.

LinearAlgebra.BLAS.scal – Function.

```
scal(n, a, X, incx)
```

Return `X` scaled by `a` for the first `n` elements of array `X` with stride `incx`.

LinearAlgebra.BLAS.iamax – Function.

```
iamax(n, dx, incx)
iamax(dx)
```

Find the index of the element of `dx` with the maximum absolute value. `n` is the length of `dx`, and `incx` is the stride. If `n` and `incx` are not provided, they assume default values of `n=length(dx)` and `incx=stride1(dx)`.

LinearAlgebra.BLAS.ger! – Function.

```
ger!(alpha, x, y, A)
```

Rank-1 update of the matrix `A` with vectors `x` and `y` as `alpha*x*y' + A`.

LinearAlgebra.BLAS.syr! – Function.

```
syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix `A` with vector `x` as `alpha*x*transpose(x) + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

LinearAlgebra.BLAS.syrk! – Function.

```
syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix `C` as `alpha*A*transpose(A) + beta*C` or `alpha*transpose(A)*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Returns `C`.

LinearAlgebra.BLAS.syrk – Function.

```
syrk(uplo, trans, alpha, A)
```

Returns either the upper triangle or the lower triangle of `A`, according to `uplo`, of `alpha*A*transpose(A)` or `alpha*transpose(A)*A`, according to `trans`.

LinearAlgebra.BLAS.her! – Function.

```
her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix `A` with vector `x` as `alpha*x*x' + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

LinearAlgebra.BLAS.herk! – Function.

```
herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix `C` as `alpha*A*A' + beta*C` or `alpha*A'*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is updated. Returns `C`.

`LinearAlgebra.BLAS.herk` – Function.

```
herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the `uplo` triangle of `alpha*A*A'` or `alpha*A'*A`, according to `trans`.

`LinearAlgebra.BLAS.gbmv!` – Function.

```
gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector `y` as `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.gbmv` – Function.

```
gbmv(trans, m, kl, ku, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'*x` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals, and `alpha` is a scalar.

`LinearAlgebra.BLAS.sbmv!` – Function.

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector `y` as `alpha*A*x + beta*y` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. The storage layout for `A` is described the reference BLAS module, level-2 BLAS at http://www.netlib.org/lapack/explore-html/. Only the `uplo` triangle of `A` is used.

Return the updated `y`.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, alpha, A, x)
```

Return `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, A, x)
```

Return `A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.gemm!` – Function.

```
gemm!(tA, tB, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` and `tB`. Return the updated `C`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, alpha, A, B)
```

Return `alpha*A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, A, B)
```

Return `A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemv!` – Function.

```
gemv!(tA, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y` or `alpha*A'x + beta*y` according to `tA`. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.gemv` – Method.

```
gemv(tA, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'x` according to `tA`. `alpha` is a scalar.

`LinearAlgebra.BLAS.gemv` – Method.

```
gemv(tA, A, x)
```

Return `A*x` or `A'x` according to `tA`.

`LinearAlgebra.BLAS.symm!` – Function.

```
symm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Return the updated `C`.

`LinearAlgebra.BLAS.symm` – Method.

```
symm(side, ul, alpha, A, B)
```

Return `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.symm` – Method.

```
symm(side, ul, A, B)
```

Return `A*B` or `B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.symv!` – Function.

```
symv!(ul, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.symv` – Method.

```
symv(ul, alpha, A, x)
```

Return `alpha*A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` is a scalar.

`LinearAlgebra.BLAS.symv` – Method.

```
symv(ul, A, x)
```

Return `A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.trmm!` – Function.

```
trmm!(side, ul, tA, dA, alpha, A, B)
```

Update `B` as `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

LinearAlgebra.BLAS.trmm – Function.

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Returns `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsm! – Function.

```
trsm!(side, ul, tA, dA, alpha, A, B)
```

Overwrite `B` with the solution to `A*X = alpha*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

LinearAlgebra.BLAS.trsm – Function.

```
trsm(side, ul, tA, dA, alpha, A, B)
```

Return the solution to `A*X = alpha*B` or one of the other three variants determined by determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trmv! – Function.

```
trmv!(ul, tA, dA, A, b)
```

Return `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on `b`.

LinearAlgebra.BLAS.trmv – Function.

```
trmv(ul, tA, dA, A, b)
```

Return `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsv! – Function.

```
trsv!(ul, tA, dA, A, b)
```

Overwrite `b` with the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones. Return the updated `b`.

LinearAlgebra.BLAS.trsv – Function.

```
trsv(ul, tA, dA, A, b)
```

Return the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.set_num_threads – Function.

```
set_num_threads(n)
```

Set the number of threads the BLAS library should use.

LinearAlgebra.I – Constant.

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Examples

```
julia> fill(1, (5,6)) * I == fill(1, (5,6))
true

julia> [1 2im 3; 1im 2 3] * I
2×3 Array{Complex{Int64},2}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

77.6 LAPACK Functions

`LinearAlgebra.LAPACK` provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in `'!'`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `ComplexF64` and `ComplexF32` arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

LinearAlgebra.LAPACK – Module.

Interfaces to LAPACK subroutines.

LinearAlgebra.LAPACK.gbtrf! – Function.

```
gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix `AB`. `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix `AB`. Returns the LU factorization in-place and `ipiv`, the vector of pivots used.

LinearAlgebra.LAPACK.gbtrs! – Function.

```
gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation `AB * X = B`. `trans` determines the orientation of `AB`. It may be `N` (no transpose), `T` (transpose), or `C` (conjugate transpose). `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix `AB`. `ipiv` is the vector of pivots returned from `gbtrf!`. Returns the vector or matrix `X`, overwriting `B` in-place.

LinearAlgebra.LAPACK.gebal! – Function.

```
gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix `A` before computing its eigensystem or Schur factorization. `job` can be one of `N` (`A` will not be permuted or scaled), `P` (`A` will only be permuted), `S` (`A` will only be scaled), or `B` (`A` will be both permuted and scaled). Modifies `A` in-place and returns `ilo`, `ihi`, and `scale`. If permuting was turned on, `A[i,j] = 0` if `j > i` and `1 < j < ilo` or `j > ihi`. `scale` contains information about the scaling/permutations performed.

LinearAlgebra.LAPACK.gebak! – Function.

```
gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors `V` of a matrix balanced using `gebal!` to the unscaled/unpermuted eigenvectors of the original matrix. Modifies `V` in-place. `side` can be `L` (left eigenvectors are transformed) or `R` (right eigenvectors are transformed).

LinearAlgebra.LAPACK.gebrd! – Function.

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce `A` in-place to bidiagonal form `A = QBP'`. Returns `A`, containing the bidiagonal matrix `B`; `d`, containing the diagonal elements of `B`; `e`, containing the off-diagonal elements of `B`; `tauq`, containing the elementary reflectors representing `Q`; and `taup`, containing the elementary reflectors representing `P`.

LinearAlgebra.LAPACK.gelqf! – Function.

```
gelqf!(A, tau)
```

Compute the LQ factorization of A, A = LQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
gelqf!(A) -> (A, tau)
```

Compute the LQ factorization of A, A = LQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqlf! – Function.

```
geqlf!(A, tau)
```

Compute the QL factorization of A, A = QL. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
geqlf!(A) -> (A, tau)
```

Compute the QL factorization of A, A = QL.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrf! – Function.

```
geqrf!(A, tau)
```

Compute the QR factorization of A, A = QR. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
geqrf!(A) -> (A, tau)
```

Compute the QR factorization of A, A = QR.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqp3! — Function.

```
geqp3!(A, [jpvt, tau]) -> (A, tau, jpvt)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. tau stores the elementary reflectors. The arguments jpvt and tau are optional and allow for passing preallocated arrays. When passed, jpvt must have length greater than or equal to n if A is an (m x n) matrix and tau must have length greater than or equal to the smallest dimension of A.

A, jpvt, and tau are modified in-place.

LinearAlgebra.LAPACK.gerqf! — Function.

```
gerqf!(A, tau)
```

Compute the RQ factorization of A, A = RQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
gerqf!(A) -> (A, tau)
```

Compute the RQ factorization of A, A = RQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrt! — Function.

```
geqrt!(A, T)
```

Compute the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
geqrt!(A, nb) -> (A, T)
```

Compute the blocked QR factorization of A, A = QR. nb sets the block size and it must be between 1 and n, the second dimension of A.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrt3! – Function.

```
geqrt3!(A, T)
```

Recursively computes the blocked `QR` factorization of `A`, `A = QR`. `T` contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of `T` sets the block size and it must be between 1 and `n`. The second dimension of `T` must equal the smallest dimension of `A`.

Returns `A` and `T` modified in-place.

```
geqrt3!(A) -> (A, T)
```

Recursively computes the blocked `QR` factorization of `A`, `A = QR`.

Returns `A`, modified in-place, and `T`, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.getrf! – Function.

```
getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted `LU` factorization of `A`, `A = LU`.

Returns `A`, modified in-place, `ipiv`, the pivoting information, and an `info` code which indicates success (`info = 0`), a singular value in `U` (`info = i`, in which case `U[i,i]` is singular), or an error code (`info < 0`).

LinearAlgebra.LAPACK.tzrzf! – Function.

```
tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix `A` to upper triangular form in-place. Returns `A` and `tau`, the scalar parameters for the elementary reflectors of the transformation.

LinearAlgebra.LAPACK.ormrz! – Function.

```
ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix `C` by `Q` from the transformation supplied by `tzrzf!`. Depending on `side` or `trans` the multiplication can be left-sided (`side = L, Q*C`) or right-sided (`side = R, C*Q`) and `Q` can be unmodified (`trans = N`), transposed (`trans = T`), or conjugate transposed (`trans = C`). Returns matrix `C` which is modified in-place with the result of the multiplication.

LinearAlgebra.LAPACK.gels! – Function.

```
gels!(trans, A, B) -> (F, B, ssr)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` using a QR or LQ factorization. Modifies the matrix/vector `B` in place with the solution. `A` is overwritten with its `QR` or `LQ` factorization. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose). `gels!` searches for the minimum norm/least squares solution. `A` may be under or over determined. The solution is returned in `B`.

`LinearAlgebra.LAPACK.gesv!` – Function.

```
gesv!(A, B) -> (B, A, ipiv)
```

Solves the linear equation `A * X = B` where `A` is a square matrix using the `LU` factorization of `A`. `A` is overwritten with its `LU` factorization and `B` is overwritten with the solution `X`. `ipiv` contains the pivoting information for the `LU` factorization of `A`.

`LinearAlgebra.LAPACK.getrs!` – Function.

```
getrs!(trans, A, ipiv, B)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` for square `A`. Modifies the matrix/vector `B` in place with the solution. `A` is the `LU` factorization from `getrf!`, with `ipiv` the pivoting information. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose).

`LinearAlgebra.LAPACK.getri!` – Function.

```
getri!(A, ipiv)
```

Computes the inverse of `A`, using its `LU` factorization found by `getrf!`. `ipiv` is the pivot information output and `A` contains the `LU` factorization of `getrf!`. `A` is overwritten with its inverse.

`LinearAlgebra.LAPACK.gesvx!` – Function.

```
gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed, R, C, B, rcond, ferr, berr, work)
```

Solves the linear equation `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) using the `LU` factorization of `A`. `fact` may be `E`, in which case `A` will be equilibrated and copied to `AF`; `F`, in which case `AF` and `ipiv` from a previous `LU` factorization are inputs; or `N`, in which case `A` will be copied to `AF` and then factored. If `fact = F`, `equed` may be `N`, meaning `A` has not been equilibrated; `R`, meaning `A` was multiplied by `Diagonal(R)` from the left; `C`, meaning `A` was multiplied by `Diagonal(C)` from the right; or `B`, meaning `A` was multiplied by `Diagonal(R)` from the left and `Diagonal(C)` from the right. If `fact = F` and `equed = R` or `B` the elements of `R` must all be positive. If `fact = F` and `equed = C` or `B` the elements of `C` must all be positive.

Returns the solution `X`; `equed`, which is an output if `fact` is not `N`, and describes the equilibration that was performed; `R`, the row equilibration diagonal; `C`, the column equilibration diagonal; `B`, which may be overwritten with its equilibrated form `Diagonal(R)*B` (if `trans = N` and `equed = R,B`) or `Diagonal(C)*B` (if `trans = T,C` and `equed = C,B`); `rcond`, the reciprocal condition number of `A` after equilbrating; `ferr`, the forward error bound for each solution vector in `X`; `berr`, the forward error bound for each solution vector in `X`; and `work`, the reciprocal pivot growth factor.

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of `gesvx!`.

LinearAlgebra.LAPACK.gelsd! – Function.

```
gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the `SVD` factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

LinearAlgebra.LAPACK.gelsy! – Function.

```
gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the full `QR` factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

LinearAlgebra.LAPACK.gglse! – Function.

```
gglse!(A, c, B, d) -> (X,res)
```

Solves the equation `A * x = c` where `x` is subject to the equality constraint `B * x = d`. Uses the formula `¦¦c - A*x¦¦^2 = 0` to solve. Returns `X` and the residual sum-of-squares.

LinearAlgebra.LAPACK.geev! – Function.

```
geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of `A`. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. Returns the eigenvalues in `W`, the right eigenvectors in `VR`, and the left eigenvectors in `VL`.

LinearAlgebra.LAPACK.gesdd! – Function.

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`, using a divide and conquer approach. If `job = A`, all the columns of `U` and the rows of `V'` are computed. If `job = N`, no columns of `U` or rows of `V'` are computed. If `job = O`, `A` is overwritten with the columns of (thin) `U` and the rows of (thin) `V'`. If `job = S`, the columns of (thin) `U` and the rows of (thin) `V'` are computed and returned separately.

LinearAlgebra.LAPACK.gesvd! – Function.

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`. If `jobu = A`, all the columns of `U` are computed. If `jobvt = A` all the rows of `V'` are computed. If `jobu = N`, no columns of `U` are computed. If `jobvt = N` no rows of `V'` are computed. If `jobu = O`, `A` is overwritten with the columns of (thin) `U`. If `jobvt = O`, `A` is overwritten with the rows of (thin) `V'`. If `jobu = S`, the columns of (thin) `U` are computed and returned separately. If `jobvt = S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `O`.

Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

LinearAlgebra.LAPACK.ggsvd! – Function.

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of `A` and `B`, `U'*A*Q = D1*R` and `V'*B*Q = D2*R`. `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu = U`, the orthogonal/unitary matrix `U` is computed. If `jobv = V` the orthogonal/unitary matrix `V` is computed. If `jobq = Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv` or `jobq` is `N`, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

LinearAlgebra.LAPACK.ggsvd3! – Function.

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of `A` and `B`, `U'*A*Q = D1*R` and `V'*B*Q = D2*R`. `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu = U`, the orthogonal/unitary matrix `U` is computed. If `jobv = V` the orthogonal/unitary matrix `V` is computed. If `jobq = Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv`, or `jobq` is `N`, that matrix is not computed. This function requires LAPACK 3.6.0.

LinearAlgebra.LAPACK.geevx! – Function.

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm, rconde, rcondv)
```

Finds the eigensystem of `A` with matrix balancing. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. If `balanc = N`, no balancing is performed. If `balanc = P`, `A` is permuted but not scaled. If `balanc = S`, `A` is scaled but not permuted. If `balanc = B`, `A` is permuted and scaled. If `sense = N`, no reciprocal condition numbers are computed. If `sense = E`, reciprocal condition numbers are computed for the eigenvalues only. If `sense = V`, reciprocal condition numbers are computed for the right eigenvectors only. If `sense = B`, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If `sense = E,B`, the right and left eigenvectors must be computed.

`LinearAlgebra.LAPACK.ggev!` — Function.

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of `A` and `B`. If `jobvl = N`, the left eigenvectors aren't computed. If `jobvr = N`, the right eigenvectors aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed.

`LinearAlgebra.LAPACK.gtsv!` — Function.

```
gtsv!(dl, d, du, B)
```

Solves the equation `A * X = B` where `A` is a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Overwrites `B` with the solution `X` and returns it.

`LinearAlgebra.LAPACK.gttrf!` — Function.

```
gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the `LU` factorization of a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Modifies `dl`, `d`, and `du` in-place and returns them and the second superdiagonal `du2` and the pivoting vector `ipiv`.

`LinearAlgebra.LAPACK.gttrs!` — Function.

```
gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) using the `LU` factorization computed by `gttrf!`. `B` is overwritten with the solution `X`.

LinearAlgebra.LAPACK.orglq! – Function.

```
orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `LQ` factorization after calling `gelqf!` on `A`. Uses the output of `gelqf!`. `A` is overwritten by `Q`.

LinearAlgebra.LAPACK.orgqr! – Function.

```
orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QR` factorization after calling `geqrf!` on `A`. Uses the output of `geqrf!`. `A` is overwritten by `Q`.

LinearAlgebra.LAPACK.orgql! – Function.

```
orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QL` factorization after calling `geqlf!` on `A`. Uses the output of `geqlf!`. `A` is overwritten by `Q`.

LinearAlgebra.LAPACK.orgrq! – Function.

```
orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `RQ` factorization after calling `gerqf!` on `A`. Uses the output of `gerqf!`. `A` is overwritten by `Q`.

LinearAlgebra.LAPACK.ormlq! – Function.

```
ormlq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `LQ` factorization of `A` computed using `gelqf!`. `C` is overwritten.

LinearAlgebra.LAPACK.ormqr! – Function.

```
ormqr!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrf!`. `C` is overwritten.

LinearAlgebra.LAPACK.ormql! – Function.

```
ormql!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QL` factorization of `A` computed using `geqlf!`. `C` is overwritten.

LinearAlgebra.LAPACK.ormrq! – Function.

```
ormrq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `RQ` factorization of `A` computed using `gerqf!`. `C` is overwritten.

LinearAlgebra.LAPACK.gemqrt! – Function.

```
gemqrt!(side, trans, V, T, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrt!`. `C` is overwritten.

LinearAlgebra.LAPACK.posv! – Function.

```
posv!(uplo, A, B) -> (A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix. If `uplo = U` the upper Cholesky decomposition of `A` is computed. If `uplo = L` the lower Cholesky decomposition of `A` is computed. `A` is overwritten by its Cholesky decomposition. `B` is overwritten with the solution `X`.

LinearAlgebra.LAPACK.potrf! – Function.

```
potrf!(uplo, A)
```

Computes the Cholesky (upper if `uplo = U`, lower if `uplo = L`) decomposition of positive-definite matrix `A`. `A` is overwritten and returned with an info code.

LinearAlgebra.LAPACK.potri! – Function.

```
potri!(uplo, A)
```

Computes the inverse of positive-definite matrix `A` after calling `potrf!` to find its (upper if `uplo = U`, lower if `uplo = L`) Cholesky decomposition.

`A` is overwritten by its inverse and returned.

`LinearAlgebra.LAPACK.potrs!` — Function.

```
potrs!(uplo, A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `potrf!`. If `uplo = U` the upper Cholesky decomposition of `A` was computed. If `uplo = L` the lower Cholesky decomposition of `A` was computed. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.pstrf!` — Function.

```
pstrf!(uplo, A, tol) -> (A, piv, rank, info)
```

Computes the (upper if `uplo = U`, lower if `uplo = L`) pivoted Cholesky decomposition of positive-definite matrix `A` with a user-set tolerance `tol`. `A` is overwritten by its Cholesky decomposition.

Returns `A`, the pivots `piv`, the rank of `A`, and an `info` code. If `info = 0`, the factorization succeeded. If `info = i > 0`, then `A` is indefinite or rank-deficient.

`LinearAlgebra.LAPACK.ptsv!` — Function.

```
ptsv!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A`. `D` is the diagonal of `A` and `E` is the off-diagonal. `B` is overwritten with the solution `X` and returned.

`LinearAlgebra.LAPACK.pttrf!` — Function.

```
pttrf!(D, E)
```

Computes the LDLt factorization of a positive-definite tridiagonal matrix with `D` as diagonal and `E` as off-diagonal. `D` and `E` are overwritten and returned.

`LinearAlgebra.LAPACK.pttrs!` — Function.

```
pttrs!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A` with diagonal `D` and off-diagonal `E` after computing `A`'s LDLt factorization using `pttrf!`. `B` is overwritten with the solution `X`.

LinearAlgebra.LAPACK.trtri! – Function.

```
trtri!(uplo, diag, A)
```

Finds the inverse of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `A` is overwritten with its inverse.

LinearAlgebra.LAPACK.trtrs! – Function.

```
trtrs!(uplo, trans, diag, A, B)
```

Solves `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) for (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `B` is overwritten with the solution `X`.

LinearAlgebra.LAPACK.trcon! – Function.

```
trcon!(norm, uplo, diag, A)
```

Finds the reciprocal condition number of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. If `norm = I`, the condition number is found in the infinity norm. If `norm = O` or `1`, the condition number is found in the one norm.

LinearAlgebra.LAPACK.trevc! – Function.

```
trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix `T`. If `side = R`, the right eigenvectors are computed. If `side = L`, the left eigenvectors are computed. If `side = B`, both sets are computed. If `howmny = A`, all eigenvectors are found. If `howmny = B`, all eigenvectors are found and backtransformed using `VL` and `VR`. If `howmny = S`, only the eigenvectors corresponding to the values in `select` are computed.

LinearAlgebra.LAPACK.trrfs! – Function.

```
trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), `adjoint(A) * X = B` (`trans = C`) for `side = L`, or the equivalent equations a right-handed `side = R` `X * A` after computing `X` using `trtrs!`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, `A` is lower triangular. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `Ferr` and `Berr` are optional inputs. `Ferr` is the forward error and `Berr` is the backward error, each component-wise.

LinearAlgebra.LAPACK.stev! – Function.

```
stev!(job, dv, ev) -> (dv, Zmat)
```

Computes the eigensystem for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `job = N` only the eigenvalues are found and returned in `dv`. If `job = V` then the eigenvectors are also found and returned in `Zmat`.

LinearAlgebra.LAPACK.stebz! – Function.

```
stebz!(range, order, vl, vu, il, iu, abstol, dv, ev) -> (dv, iblock, isplit)
```

Computes the eigenvalues for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. If `order = B`, eigvalues are ordered within a block. If `order = E`, they are ordered across all the blocks. `abstol` can be set as a tolerance for convergence.

LinearAlgebra.LAPACK.stegr! – Function.

```
stegr!(jobz, range, dv, ev, vl, vu, il, iu) -> (w, Z)
```

Computes the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. The eigenvalues are returned in `w` and the eigenvectors in `Z`.

LinearAlgebra.LAPACK.stein! – Function.

```
stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

LinearAlgebra.LAPACK.syconv! – Function.

```
syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

LinearAlgebra.LAPACK.sysv! – Function.

```
sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

LinearAlgebra.LAPACK.sytrf! – Function.

```
sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

LinearAlgebra.LAPACK.sytri! – Function.

```
sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

LinearAlgebra.LAPACK.sytrs! – Function.

```
sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

LinearAlgebra.LAPACK.hesv! – Function.

```
hesv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

LinearAlgebra.LAPACK.hetrf! – Function.

```
hetrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

LinearAlgebra.LAPACK.hetri! – Function.

```
hetri!(uplo, A, ipiv)
```

Computes the inverse of a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

LinearAlgebra.LAPACK.hetrs! – Function.

```
hetrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

LinearAlgebra.LAPACK.syev! – Function.

```
syev!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

LinearAlgebra.LAPACK.syevr! – Function.

```
syevr!(jobz, range, uplo, A, vl, vu, il, iu, abstol) -> (W, Z)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

LinearAlgebra.LAPACK.sygvd! – Function.

```
sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)
```

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A` and symmetric positive-definite matrix `B`. If `uplo = U`, the upper triangles of `A` and `B` are used. If `uplo = L`, the lower triangles of `A` and `B` are used. If `itype = 1`, the problem to solve is `A * x = lambda * B * x`. If `itype = 2`, the problem to solve is `A * B * x = lambda * x`. If `itype = 3`, the problem to solve is `B * A * x = lambda * x`.

LinearAlgebra.LAPACK.bdsqr! – Function.

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product `Q' * C`.

Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

LinearAlgebra.LAPACK.bdsdc! – Function.

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal using a divide and conqueq method. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. If `compq = N`, only the singular values are found. If `compq = I`, the singular values and vectors are found. If `compq = P`, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in `d`, and if `compq = P`, the compact singular vectors in `iq`.

LinearAlgebra.LAPACK.gecon! – Function.

```
gecon!(normtype, A, anorm)
```

Finds the reciprocal condition number of matrix `A`. If `normtype = I`, the condition number is found in the infinity norm. If `normtype = O` or `1`, the condition number is found in the one norm. `A` must be the result of `getrf!` and `anorm` is the norm of `A` in the relevant norm.

LinearAlgebra.LAPACK.gehrd! – Function.

```
gehrd!(ilo, ihi, A) -> (A, tau)
```

Converts a matrix `A` to Hessenberg form. If `A` is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo = 1` and `ihi = size(A,2)`. `tau` contains the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.orghr! – Function.

```
orghr!(ilo, ihi, A, tau)
```

Explicitly finds `Q`, the orthogonal/unitary matrix from `gehrd!`. `ilo`, `ihi`, `A`, and `tau` must correspond to the input/output to `gehrd!`.

LinearAlgebra.LAPACK.gees! – Function.

```
gees!(jobvs, A) -> (A, vs, w)
```

Computes the eigenvalues (`jobvs = N`) or the eigenvalues and Schur vectors (`jobvs = V`) of matrix `A`. `A` is overwritten by its Schur form.

Returns `A`, `vs` containing the Schur vectors, and `w`, containing the eigenvalues.

LinearAlgebra.LAPACK.gges! – Function.

```
gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B`.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

LinearAlgebra.LAPACK.trexc! – Function.

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization of a matrix. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. `ifst` and `ilst` specify the reordering of the vectors.

LinearAlgebra.LAPACK.trsen! – Function.

```
trsen!(compq, job, select, T, Q) -> (T, Q, w, s, sep)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If `job = N`, no condition numbers are found. If `job = E`, only the condition number for this cluster of eigenvalues is found. If `job = V`, only the condition number for the invariant subspace is found. If `job = B` then the condition numbers for the cluster and subspace are found. If `compq = V` the Schur vectors `Q` are updated. If `compq = N` the Schur vectors are not modified. `select` determines which eigenvalues are in the cluster.

Returns `T`, `Q`, reordered eigenvalues in `w`, the condition number of the cluster of eigenvalues `s`, and the condition number of the invariant subspace `sep`.

LinearAlgebra.LAPACK.tgsen! – Function.

```
tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. `select` specifies the eigenvalues in each cluster.

LinearAlgebra.LAPACK.trsyl! – Function.

```
trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation `A * X +/- X * B = scale*C` where `A` and `B` are both quasi-upper triangular. If `transa = N`, `A` is not modified. If `transa = T`, `A` is transposed. If `transa = C`, `A` is conjugate transposed. Similarly for `transb` and `B`. If `isgn = 1`, the equation `A * X + X * B = scale * C` is solved. If `isgn = -1`, the equation `A * X - X * B = scale * C` is solved.

Returns `X` (overwriting `C`) and `scale`.

Chapter 78

Logging

The `Logging` module provides a way to record the history and progress of a computation as a log of events. Events are created by inserting a logging statement into the source code, for example:

```
@warn "Abandon printf debugging, all ye who enter here!"
```

```
┌ Warning: Abandon printf debugging, all ye who enter here!
└ @ Main REPL[1]:1
```

The system provides several advantages over peppering your source code with calls to `println()`. First, it allows you to control the visibility and presentation of messages without editing the source code. For example, in contrast to the `@warn` above

```
@debug "The sum of some values $(sum(rand(100)))"
```

will produce no output by default. Furthermore, it's very cheap to leave debug statements like this in the source code because the system avoids evaluating the message if it would later be ignored. In this case `sum(rand(100))` and the associated string processing will never be executed unless debug logging is enabled.

Second, the logging tools allow you to attach arbitrary data to each event as a set of key–value pairs. This allows you to capture local variables and other program state for later analysis. For example, to attach the local array variable `A` and the sum of a vector `v` as the key `s` you can use

```
A = ones(Int, 4, 4)
v = ones(100)
@info "Some variables"  A  s=sum(v)


# output
```

```
┌ Info: Some variables
│   A =
│    4×4 Array{Int64,2}:
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
└   s = 100.0
```

All of the logging macros **`@debug`**, **`@info`**, **`@warn`** and **`@error`** share common features that are described in detail in the documentation for the more general macro `@logmsg`.

78.1 Log event structure

Each event generates several pieces of data, some provided by the user and some automatically extracted. Let's examine the user-defined data first:

- The log level is a broad category for the message that is used for early filtering. There are several standard levels of type `LogLevel`; user-defined levels are also possible.
  - Use **`Debug`** for verbose information that could be useful when debugging an application or module. These events are disabled by default.
  - Use **`Info`** to inform the user about the normal operation of the program.
  - Use **`Warn`** when a potential problem is detected.
  - Use **`Error`** to report errors where the code has enough context to recover and continue. (When the code doesn't have enough context, an exception or early return is more appropriate.)
- The message is an object describing the event. By convention **`AbstractString`**s passed as messages are assumed to be in markdown format. Other types will be displayed using **`show(io,mime,obj)`** according to the display capabilities of the installed logger.
- Optional key–value pairs allow arbitrary data to be attached to each event. Some keys have conventional meaning that can affect the way an event is interpreted (see `@logmsg`).

The system also generates some standard information for each event:

- The **`module`** in which the logging macro was expanded.
- The **`file`** and **`line`** where the logging macro occurs in the source code.

- A message `id` that is a unique, fixed identifier for the source code statement where the logging macro appears. This identifier is designed to be fairly stable even if the source code of the file changes, as long as the logging statement itself remains the same.
- A `group` for the event, which is set to the base name of the file by default, without extension. This can be used to group messages into categories more finely than the log level (for example, all deprecation warnings have group `:depwarn`), or into logical groupings across or within modules.

Notice that some useful information such as the event time is not included by default. This is because such information can be expensive to extract and is also dynamically available to the current logger. It's simple to define a custom logger to augment event data with the time, backtrace, values of global variables and other useful information as required.

78.2 Processing log events

As you can see in the examples, logging statements make no mention of where log events go or how they are processed. This is a key design feature that makes the system composable and natural for concurrent use. It does this by separating two different concerns:

- Creating log events is the concern of the module author who needs to decide where events are triggered and which information to include.
- Processing of log events — that is, display, filtering, aggregation and recording — is the concern of the application author who needs to bring multiple modules together into a cooperating application.

Loggers

Processing of events is performed by a logger, which is the first piece of user configurable code to see the event. All loggers must be subtypes of `AbstractLogger`.

When an event is triggered, the appropriate logger is found by looking for a task-local logger with the global logger as fallback. The idea here is that the application code knows how log events should be processed and exists somewhere at the top of the call stack. So we should look up through the call stack to discover the logger — that is, the logger should be dynamically scoped. (This is a point of contrast with logging frameworks where the logger is lexically scoped; provided explicitly by the module author or as a simple global variable. In such a system it's awkward to control logging while composing functionality from multiple modules.)

The global logger may be set with `global_logger`, and task-local loggers controlled using `with_logger`. Newly spawned tasks inherit the logger of the parent task.

There are three logger types provided by the library. `ConsoleLogger` is the default logger you see when starting the REPL. It displays events in a readable text format and tries to give simple but user friendly control over formatting and filtering. `NullLogger` is a convenient way to drop all messages where necessary; it is the logging equivalent of the `devnull` stream. `SimpleLogger` is a very simplistic text formatting logger, mainly useful for debugging the logging system itself.

Custom loggers should come with overloads for the functions described in the reference section.

Early filtering and message handling

When an event occurs, a few steps of early filtering occur to avoid generating messages that will be discarded:

1. The message log level is checked against a global minimum level (set via `disable_logging`). This is a crude but extremely cheap global setting.

2. The current logger state is looked up and the message level checked against the logger's cached minimum level, as found by calling `Logging.min_enabled_level`. This behavior can be overridden via environment variables (more on this later).

3. The `Logging.shouldlog` function is called with the current logger, taking some minimal information (level, module, group, id) which can be computed statically. Most usefully, `shouldlog` is passed an event `id` which can be used to discard events early based on a cached predicate.

If all these checks pass, the message and key–value pairs are evaluated in full and passed to the current logger via the `Logging.handle_message` function. `handle_message()` may perform additional filtering as required and display the event to the screen, save it to a file, etc.

Exceptions that occur while generating the log event are captured and logged by default. This prevents individual broken events from crashing the application, which is helpful when enabling little-used debug events in a production system. This behavior can be customized per logger type by extending `Logging.catch_exceptions`.

78.3 Testing log events

Log events are a side effect of running normal code, but you might find yourself wanting to test particular informational messages and warnings. The `Test` module provides a `@test_logs` macro that can be used to pattern match against the log event stream.

78.4 Environment variables

Message filtering can be influenced through the `JULIA_DEBUG` environment variable, and serves as an easy way to enable debug logging for a file or module. For example, loading julia with `JULIA_DEBUG=loading` will activate `@debug` log messages in `loading.jl`:

```
$ JULIA_DEBUG=loading julia -e 'using OhMyREPL'
 Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji due to it containing an invalid cache
     header
 @ Base loading.jl:1328
[ Info: Recompiling stale cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji for module OhMyREPL
 Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/Tokenize.ji due to it containing an invalid cache
     header
 @ Base loading.jl:1328
...
```

Similarly, the environment variable can be used to enable debug logging of modules, such as `Pkg`, or module roots (see `Base.moduleroot`). To enable all debug logging, use the special value `all`.

78.5 Writing log events to a file

Sometimes it can be useful to write log events to a file. Here is an example of how to use a task-local and global logger to write information to a text file:

```
# Load the logging module
julia> using Logging

# Open a textfile for writing
julia> io = open("log.txt", "w+")
IOStream(<file log.txt>)

# Create a simple logger
julia> logger = SimpleLogger(io)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# Log a task-specific message
julia> with_logger(logger) do
           @info("a context specific log message")
       end
```

```
# Write all buffered messages to the file
julia> flush(io)

# Set the global logger to logger
julia> global_logger(logger)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# This message will now also be written to the file
julia> @info("a global log message")

# Close the file
julia> close(io)
```

78.6 Reference

Logging module

`Logging.Logging` – Module.

Utilities for capturing, filtering and presenting streams of log events. Normally you don't need to import `Logging` to create log events; for this the standard logging macros such as `@info` are already exported by `Base` and available by default.

Creating events

`Logging.@logmsg` – Macro.

```
@debug message  [key=value | value ...]
@info  message  [key=value | value ...]
@warn  message  [key=value | value ...]
@error message  [key=value | value ...]

@logmsg level message [key=value | value ...]
```

Create a log record with an informational `message`. For convenience, four logging macros `@debug`, `@info`, `@warn` and `@error` are defined which log at the standard severity levels `Debug`, `Info`, `Warn` and `Error`. `@logmsg` allows `level` to be set programmatically to any `LogLevel` or custom log level types.

`message` should be an expression which evaluates to a string which is a human readable description of the log event. By convention, this string will be formatted as markdown when presented.

The optional list of `key=value` pairs supports arbitrary user defined metadata which will be passed through to the logging backend as part of the log record. If only a `value` expression is supplied, a key representing the expression will be generated using `Symbol`. For example, `x` becomes `x=x`, and `foo(10)` becomes `Symbol("foo(10)")=foo(10)`. For splatting a list of key value pairs, use the normal splatting syntax, `@info "blah" kws...`.

There are some keys which allow automatically generated log data to be overridden:

- `_module=mod` can be used to specify a different originating module from the source location of the message.
- `_group=symbol` can be used to override the message group (this is normally derived from the base name of the source file).
- `_id=symbol` can be used to override the automatically generated unique message identifier. This is useful if you need to very closely associate messages generated on different source lines.
- `_file=string` and `_line=integer` can be used to override the apparent source location of a log message.

There's also some key value pairs which have conventional meaning:

- `maxlog=integer` should be used as a hint to the backend that the message should be displayed no more than `maxlog` times.
- `exception=ex` should be used to transport an exception with a log message, often used with `@error`. An associated backtrace `bt` may be attached using the tuple `exception=(ex,bt)`.

Examples

```
@debug "Verbose debugging information.  Invisible by default"
@info  "An informational message"
@warn  "Something was odd.  You should pay attention"
@error "A non fatal error occurred"

x = 10
@info "Some variables attached to the message" x a=42.0

@debug begin
    sA = sum(A)
    "sum(A) = $sA is an expensive operation, evaluated only when `shouldlog` returns true"
end

for i=1:10000
    @info "With the default backend, you will only see (i = $i) ten times"  maxlog=10
    @debug "Algorithm1" i progress=i/10000
end
```

source

Logging.LogLevel – Type.

```
LogLevel(level)
```

Severity/verbosity of a log record.

The log level provides a key against which potential log records may be filtered, before any other work is done to construct the log record data structure itself.

source

Processing events with AbstractLogger

Event processing is controlled by overriding functions associated with `AbstractLogger`:

Methods to implement		Brief description
Logging.handle_message		Handle a log event
Logging.shouldlog		Early filtering of events
Logging.min_enabled_level		Lower bound for log level of accepted events
Optional methods	Default definition	Brief description
Logging.catch_exceptions	`true`	Catch exceptions during event evaluation

Logging.AbstractLogger – Type.

A logger controls how log records are filtered and dispatched. When a log record is generated, the logger is the first piece of user configurable code which gets to inspect the record and decide what to do with it.

source

Logging.handle_message – Function.

```
handle_message(logger, level, message, _module, group, id, file, line; key1=val1, ...)
```

Log a message to `logger` at `level`. The logical location at which the message was generated is given by module `_module` and `group`; the source location by `file` and `line`. `id` is an arbitrary unique value (typically a Symbol) to be used as a key to identify the log statement when filtering.

source

Logging.shouldlog – Function.

```
shouldlog(logger, level, _module, group, id)
```

Return true when `logger` accepts a message at `level`, generated for `_module`, `group` and with unique log identifier `id`.

source

`Logging.min_enabled_level` – Function.

```
min_enabled_level(logger)
```

Return the minimum enabled level for `logger` for early filtering. That is, the log level below or equal to which all messages are filtered.

source

`Logging.catch_exceptions` – Function.

```
catch_exceptions(logger)
```

Return true if the logger should catch exceptions which happen during log record construction. By default, messages are caught

By default all exceptions are caught to prevent log message generation from crashing the program. This lets users confidently toggle little-used functionality - such as debug logging - in a production system.

If you want to use logging as an audit trail you should disable this for your logger type.

source

`Logging.disable_logging` – Function.

```
disable_logging(level)
```

Disable all log messages at log levels equal to or less than `level`. This is a global setting, intended to make debug logging extremely cheap when disabled.

source

Using Loggers

Logger installation and inspection:

`Logging.global_logger` – Function.

```
global_logger()
```

Return the global logger, used to receive messages when no specific logger exists for the current task.

```
global_logger(logger)
```

Set the global logger to `logger`, and return the previous global logger.

source

`Logging.with_logger` — Function.

```
with_logger(function, logger)
```

Execute `function`, directing all log messages to `logger`.

Example

```
function test(x)
    @info "x = $x"
end

with_logger(logger) do
    test(1)
    test([1,2])
end
```

source

`Logging.current_logger` — Function.

```
current_logger()
```

Return the logger for the current task, or the global logger if none is attached to the task.

source

Loggers that are supplied with the system:

`Logging.NullLogger` — Type.

```
NullLogger()
```

Logger which disables all messages and produces no output - the logger equivalent of /dev/null.

source

Logging.ConsoleLogger – Type.

```
ConsoleLogger(stream=stderr, min_level=Info; meta_formatter=default_metafmt,
              show_limited=true, right_justify=0)
```

Logger with formatting optimized for readability in a text console, for example interactive work with the Julia REPL.

Log levels less than `min_level` are filtered out.

Message formatting can be controlled by setting keyword arguments:

- `meta_formatter` is a function which takes the log event metadata `(level, _module, group, id, file, line)` and returns a color (as would be passed to printstyled), prefix and suffix for the log message. The default is to prefix with the log level and a suffix containing the module, file and line location.
- `show_limited` limits the printing of large data structures to something which can fit on the screen by setting the `:limit IOContext` key during formatting.
- `right_justify` is the integer column which log metadata is right justified at. The default is zero (metadata goes on its own line).

Logging.SimpleLogger – Type.

```
SimpleLogger(stream=stderr, min_level=Info)
```

Simplistic logger for logging all messages with level greater than or equal to `min_level` to `stream`.

source

Chapter 79

Markdown

This section describes Julia's markdown syntax, which is enabled by the Markdown standard library. The following Markdown elements are supported:

79.1 Inline elements

Here "inline" refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

Bold

Surround words with two asterisks, **, to display the enclosed text in boldface.

```
A paragraph containing a **bold** word.
```

Italics

Surround words with one asterisk, *, to display the enclosed text in italics.

```
A paragraph containing an *emphasized* word.
```

Literals

Surround text that should be displayed exactly as written with single backticks, ` .

```
A paragraph containing a `literal` word.
```

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

Tip

To include a backtick character within literal text use three backticks rather than one to enclose the text.

```
A paragraph containing a ``` `backtick` character ```.
```

By extension any odd number of backticks may be used to enclose a lesser number of backticks.

LaTeX

Surround text that should be displayed as mathematics using LaTeX syntax with double backticks, `` .

```
A paragraph containing some ``\LaTeX`` markup.
```

Tip

As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within LaTeX markup then two enclosing backticks is sufficient.

Note

The `\` character should be escaped appropriately if the text is embedded in a Julia source code, for example, `"``\\LaTeX`` syntax in a docstring."`, since it is interpreted as a string literal. Alternatively, in order to avoid escaping, it is possible to use the `raw` string macro together with the `@doc` macro:

```
@doc raw"``\LaTeX`` syntax in a docstring." functionname
```

Links

Links to either external or internal addresses can be written using the following syntax, where the text enclosed in square brackets, `[ ]`, is the name of the link and the text enclosed in parentheses, `( )`, is the URL.

```
A paragraph containing a link to [Julia](http://www.julialang.org).
```

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

```
"""
    tryparse(type, str; base)

Like [`parse`](@ref), but returns either a value of the requested type,
or [`nothing`](@ref) if the string does not contain a valid number.
"""
```

This will create a link in the generated docs to the `parse` documentation (which has more information about what this function actually does), and to the `nothing` documentation. It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

> Note
>
> The above cross referencing is not a Markdown feature, and relies on Documenter.jl, which is used to build base Julia's documentation.

Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

```
A paragraph containing a numbered footnote [^1] and a named one [^named].
```

> Note
>
> The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the Footnotes section below.

79.2 Toplevel elements

The following elements can be written either at the "toplevel" of a document or within another "toplevel" element.

Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the Inline elements section above, with one or more blank lines above and below it.

```
This is a paragraph.


And this is *another* one containing some emphasized text.
A new line, but still part of the same paragraph.
```

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

```
# Level One
## Level Two
### Level Three
```

```
#### Level Four
##### Level Five
###### Level Six
```

A header line can contain any inline syntax in the same way as a paragraph can.

> Tip
>
> Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

Code blocks

Source code can be displayed as a literal block using an indent of four spaces as shown in the following example.

```
This is a paragraph.


    function func(x)
        # ...
    end


Another paragraph.
```

Additionally, code blocks can be enclosed using triple backticks with an optional "language" to specify how a block of code should be highlighted.

````
A code block without a "language":


```
function func(x)
    # ...
end
```


and another one with the "language" specified as `julia`:


```julia
function func(x)
    # ...
end
```
````

Note

"Fenced" code blocks, as shown in the last example, should be preferred over indented code blocks since there is no way to specify what language an indented code block is written in.

Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using > characters prepended to each line of the quote as follows.

```
Here's a quote:

> Julia is a high-level, high-performance dynamic programming language for
> technical computing, with syntax that is familiar to users of other
> technical computing environments.
```

Note that a single space must appear after the > character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

Images

The syntax for images is similar to the link syntax mentioned above. Prepending a ! character to a link will display an image from the specified URL rather than a link to it.

```
![alternative text](link/to/image.png)
```

Lists

Unordered lists can be written by prepending each item in a list with either *, +, or -.

```
A list of items:

  * item one
  * item two
  * item three
```

Note the two spaces before each * and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

```
Another list:
```

```
 * item one

 * item two

   ```
   f(x) = x
   ```

 * And a sublist:

     + sub-item one
     + sub-item two
```

Note

The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the `i` in `item two`.

Ordered lists are written by replacing the "bullet" character, either `*`, `+`, or `-`, with a positive integer followed by either `.` or `)`.

```
Two ordered lists:

 1. item one
 2. item two
 3. item three

 5) item five
 6) item six
 7) item seven
```

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large LaTeXequations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `math` as in the example below.

````
```math
f(a) = \frac{1}{2\pi}\int_{0}^{2\pi} (\alpha+R\cos(\theta))d\theta
```
````

Footnotes

This syntax is paired with the inline syntax for Footnote references. Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the `:` character that is appended to the footnote label.

````
[^1]: Numbered footnote text.

[^note]:

    Named footnote text containing several toplevel elements.

      * item one
      * item two
      * item three

    ```julia
    function func(x)
        # ...
    end
    ```
````

> **Note**
>
> No checks are done during parsing to make sure that all footnote references have matching footnotes.

Horizontal rules

The equivalent of an `<hr>` HTML tag can be written using the following syntax:

```
Text above the line.

---

And text below the line.
```

Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above – only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

```
¦ Column One ¦ Column Two ¦ Column Three ¦
¦:---------- ¦ ---------- ¦:------------:¦
¦ Row `1`    ¦ Column `2` ¦              ¦
¦ *Row* 2    ¦ **Row** 2  ¦ Column ``3`` ¦
```

> Note
>
> As illustrated in the above example each column of ¦ characters must be aligned vertically.
>
> A `:` character on either end of a column's header separator (the row containing `-` characters) specifies whether the row is left-aligned, right-aligned, or (when `:` appears on both ends) center-aligned. Providing no `:` characters will default to right-aligning the column.

Admonitions

Specially formatted blocks, known as admonitions, can be used to highlight particular remarks. They can be defined using the following `!!!` syntax:

```
!!! note

    This is the content of the note.

!!! warning "Beware!"

    And this is another one.

    This warning admonition has a custom title: `"Beware!"`.
```

The type of the admonition can be any word, but some types produce special styling, namely (in order of decreasing severity): `danger`, `warning`, `info`/`note`, and `tip`.

A custom title for the box can be provided as a string (in double quotes) after the admonition type. If no title text is specified after the admonition type, then the title used will be the type of the block, i.e. `"Note"` in the case of the `note` admonition.

Admonitions, like most other toplevel elements, can contain other toplevel elements.

79.3 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown content is rendered the usual `show` methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.

Chapter 80

Memory-mapped I/O

`Mmap.Anonymous` — Type.

```
Mmap.Anonymous(name::AbstractString="", readonly::Bool=false, create::Bool=true)
```

Create an `IO`-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in `Mmap.mmap`. Used by `SharedArray` for creating shared memory arrays.

Examples

```
julia> using Mmap

julia> anon = Mmap.Anonymous();

julia> isreadable(anon)
true

julia> iswritable(anon)
true

julia> isopen(anon)
true
```

`Mmap.mmap` — Function.

```
Mmap.mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Array{T,N}}, dims, offset];
↪  grow::Bool=true, shared::Bool=true)
Mmap.mmap(type::Type{Array{T,N}}, dims)
```

Create an `Array` whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an `Array{T,N}` with a bits-type element of `T` and dimension `N` that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the stream argument, either as an open `IOStream` or filename string. When you initialize the stream, use `"r"` for a "read-only" array, and `"w+"` to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `IOStream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmapping
# (you could alternatively use mmap to do this step, too)
using Mmap
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin")   # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = Mmap.mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

```
Mmap.mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Examples

```
julia> using Mmap

julia> io = open("mmap.bin", "w+");

julia> B = Mmap.mmap(io, BitArray, (25,30000));

julia> B[3, 4000] = true;

julia> Mmap.sync!(B);

julia> close(io);

julia> io = open("mmap.bin", "r+");

julia> C = Mmap.mmap(io, BitArray, (25,30000));

julia> C[3, 4000]
true

julia> C[2, 4000]
false

julia> close(io)

julia> rm("mmap.bin")
```

This creates a 25-by-30000 `BitArray`, linked to the file associated with stream `io`.

`Mmap.sync!` – Function.

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

Chapter 81

Pkg

Pkg is Julia's builtin package manager, and handles operations such as installing, updating and removing packages.

> Note
>
> What follows is a very brief introduction to Pkg. It is highly recommended to read the full manual, which is available here: https://julialang.github.io/Pkg.jl/v1/.

What follows is a quick overview of Pkg, Julia's package manager. It should help new users become familiar with basic Pkg features.

Pkg comes with a REPL. Enter the Pkg REPL by pressing] from the Julia REPL. To get back to the Julia REPL, press backspace or ^C.

> Note
>
> This guide relies on the Pkg REPL to execute Pkg commands. For non-interactive use, we recommend the Pkg API. The Pkg API is fully documented in the API Reference section of the Pkg documentation.

Upon entering the Pkg REPL, you should see a similar prompt:

```
(v1.1) pkg>
```

To add a package, use `add`:

```
(v1.1) pkg> add Example
```

> Note
>
> Some Pkg output has been omitted in order to keep this guide focused. This will help maintain a good pace and not get bogged down in details. If you require more details, refer to subsequent sections of the Pkg manual.

We can also specify multiple packages at once:

```
(v1.1) pkg> add JSON StaticArrays
```

To remove packages, use `rm`:

```
(v1.1) pkg> rm JSON StaticArrays
```

So far, we have referred only to registered packages. Pkg also supports working with unregistered packages. To add an unregistered package, specify a URL:

```
(v1.1) pkg> add https://github.com/JuliaLang/Example.jl
```

Use `rm` to remove this package by name:

```
(v1.1) pkg> rm Example
```

Use `update` to update an installed package:

```
(v1.1) pkg> update Example
```

To update all installed packages, use `update` without any arguments:

```
(v1.1) pkg> update
```

Up to this point, we have covered basic package management: adding, updating and removing packages. This will be familiar if you have used other package managers. Pkg offers significant advantages over traditional package managers by organizing dependencies into environments.

You may have noticed the `(v1.1)` in the REPL prompt. This lets us know `v1.1` is the active environment. The active environment is the environment that will be modified by Pkg commands such as `add`, `rm` and `update`.

Let's set up a new environment so we may experiment. To set the active environment, use `activate`:

```
(v1.1) pkg> activate tutorial
[ Info: activating new environment at `/tmp/tutorial/Project.toml`.
```

Pkg lets us know we are creating a new environment and that this environment will be stored in the `/tmp/tutorial` directory.

Pkg has also updated the REPL prompt in order to reflect the new active environment:

```
(tutorial) pkg>
```

We can ask for information about the active environment by using `status`:

```
(tutorial) pkg> status
    Status `/tmp/tutorial/Project.toml`
   (empty environment)
```

`/tmp/tutorial/Project.toml` is the location of the active environment's project file. A project file is where Pkg stores metadata for an environment. Notice this new environment is empty. Let us add a package and observe:

```
(tutorial) pkg> add Example
...

(tutorial) pkg> status
    Status `/tmp/tutorial/Project.toml`
  [7876af07] Example v0.5.1
```

We can see `tutorial` now contains `Example` as a dependency.

Say we are working on `Example` and feel it needs new functionality. How can we modify the source code? We can use `develop` to set up a git clone of the `Example` package.

```
(tutorial) pkg> develop --local Example
...

(tutorial) pkg> status
    Status `/tmp/tutorial/Project.toml`
  [7876af07] Example v0.5.1+ [`dev/Example`]
```

Notice the feedback has changed. `dev/Example` refers to the location of the newly created clone. If we look inside the `/tmp/tutorial` directory, we will notice the following files:

```
tutorial├──
 dev│
   └── Example├──
 Manifest.toml└──
 Project.toml
```

Instead of loading a registered version of `Example`, Julia will load the source code contained in `tutorial/dev/Example`.

Let's try it out. First we modify the file at `tutorial/dev/Example/src/Example.jl` and add a simple function:

```
plusone(x::Int) = x + 1
```

Now we can go back to the Julia REPL and load the package:

```
julia> import Example
```

> Warn
>
> A package can only be loaded once per Julia session. If you have run `import Example` in the current Julia session, you will have to restart Julia and rerun `activate tutorial` in the Pkg REPL. Revise.jl can make this process significantly more pleasant, but setting it up is beyond the scope of this guide.

Julia should load our new code. Let's test it:

```
julia> Example.plusone(1)
2
```

Say we have a change of heart and decide the world is not ready for such elegant code. We can tell Pkg to stop using the local clone and use a registered version instead. We do this with `free`:

```
(tutorial) pkg> free Example
```

When you are done experimenting with `tutorial`, you can return to the default environment by running `activate` with no arguments:

```
(tutorial) pkg> activate

(v1.1) pkg>
```

If you are ever stuck, you can ask `Pkg` for help:

```
(v1.1) pkg> ?
```

You should see a list of available commands along with short descriptions. You can ask for more detailed help by specifying a command:

```
(v1.1) pkg> ?develop
```

This guide should help you get started with `Pkg`. `Pkg` has much more to offer in terms of powerful package management, read the full manual to learn more!

Chapter 82

Printf

Printf.@printf – Macro.

```
@printf([io::IOStream], "%Fmt", args...)
```

Print `args` using C `printf` style format specification string, with some caveats: `Inf` and `NaN` are printed consistently as `Inf` and `NaN` for flags `%a`, `%A`, `%e`, `%E`, `%f`, `%F`, `%g`, and `%G`. Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Optionally, an `IOStream` may be passed as the first argument to redirect output.

See also: `@sprintf`

Examples

```
julia> @printf("%f %F %f %F\n", Inf, Inf, NaN, NaN)
Inf Inf NaN NaN



julia> @printf "%.0f %.1f %f\n" 0.5 0.025 -0.0078125
1 0.0 -0.007813
```

Printf.@sprintf – Macro.

```
@sprintf("%Fmt", args...)
```

Return `@printf` formatted output as string.

Examples

```
julia> s = @sprintf "this is a %s %15.1f" "test" 34.567;

julia> println(s)
this is a test            34.6
```

Chapter 83

Profiling

`Profile.@profile` – Macro.

```
@profile
```

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

The methods in `Profile` are not exported and need to be called e.g. as `Profile.print()`.

`Profile.clear` – Function.

```
clear()
```

Clear any existing backtraces from the internal buffer.

`Profile.print` – Function.

```
print([io::IO = stdout,] [data::Vector]; kwargs...)
```

Prints profiling results to `io` (by default, `stdout`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

- `format` – Determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded).
- `combine` – If `true` (default), instruction pointers are merged that correspond to the same line of code.

- `maxdepth` – Limits the depth higher than `maxdepth` in the `:tree` format.
- `sortedby` – Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, `:count` sorts in order of number of collected samples, and `:overhead` sorts by the number of samples incurred by each function by itself.
- `noisefloor` – Limits frames that exceed the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.
- `mincount` – Limits the printout to only those lines with at least `mincount` occurrences.
- `recur` – Controls the recursion handling in `:tree` format. `:off` (default) prints the tree as normal. `:flat` instead compresses any recursion (by ip), showing the approximate effect of converting any self-recursion into an iterator. `:flatc` does the same but also includes collapsing of C frames (may do odd things around `jl_apply`).

```
print([io::IO = stdout,] data::Vector, lidict::LineInfoDict; kwargs...)
```

Prints profiling results to `io`. This variant is used to examine results exported by a previous call to `retrieve`. Supply the vector `data` of backtraces and a dictionary `lidict` of line information.

See `Profile.print([io], data)` for an explanation of the valid keyword arguments.

`Profile.init` – Function.

```
init(; n::Integer, delay::Real))
```

Configure the `delay` between backtraces (measured in seconds), and the number `n` of instruction pointers that may be stored. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Default settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order `(n, delay)`.

`Profile.fetch` – Function.

```
fetch() -> data
```

Returns a copy of the buffer of profile backtraces. Note that the values in `data` have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; `retrieve` may be a better choice for most users.

`Profile.retrieve` – Function.

```
retrieve() -> data, lidict
```

"Exports" profiling results in a portable format, returning the set of all backtraces (`data`) and a dictionary that maps the (session-specific) instruction pointers in `data` to `LineInfo` values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

`Profile.callers` – Function.

```
callers(funcname, [data, lidict], [filename=<filename>], [linerange=<start:stop>]) -> Vector{Tuple{count,
↪  lineinfo}}
```

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace `data` obtained from `retrieve`; otherwise, the current internal profile buffer is used.

`Profile.clear_malloc_data` – Function.

```
clear_malloc_data()
```

Clears any stored memory allocation data when running julia with `--track-allocation`. Execute the command(s) you want to test (to force JIT-compilation), then call `clear_malloc_data`. Then execute your command(s) again, quit Julia, and examine the resulting `*.mem` files.

Chapter 84

The Julia REPL

Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the `julia` executable. In addition to allowing quick and easy evaluation of Julia statements, it has a searchable history, tab-completion, many helpful keybindings, and dedicated help and shell modes. The REPL can be started by simply calling `julia` with no arguments or double-clicking on the executable:

```
$ julia


               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.3 (2020-11-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia>
```

To exit the interactive session, type `^D` – the control key together with the `d` key on a blank line – or type `exit()` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

84.1 The different prompt modes

The Julian mode

The REPL has four main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`. It is here that you can enter Julia expressions. Hitting

return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

```
julia> string(1 + 2)
"3"
```

There are a number useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable `ans`. A trailing semicolon on the line can be used as a flag to suppress showing the result.

```
julia> string(3 * 4);

julia> ans
"12"
```

In Julia mode, the REPL supports something called prompt pasting. This activates when pasting text that starts with `julia>` into the REPL. In that case, only expressions starting with `julia>` are parsed, others are removed. This makes it is possible to paste a chunk of code that has been copied from a REPL session without having to scrub away prompts and outputs. This feature is enabled by default but can be disabled or enabled at will with `REPL.enable_promptpaste(::Bool)`. If it is enabled, you can try it out by pasting the code block above this paragraph straight into the REPL. This feature does not work on the standard Windows command prompt due to its limitation at detecting when a paste occurs.

Help mode

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing `?`. Julia will attempt to print help or documentation for anything entered in help mode:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help?>

help?> string
search: string String Cstring Cwstring RevString randstring bytestring SubString

  string(xs...)

  Create a string from any values using the print function.
```

Macros, types and variables can also be queried:

```
help?> @time
  @time

  A macro to execute an expression, printing the time it took to execute, the number of allocations,
  and the total number of bytes its execution caused to be allocated, before returning the value of the
  expression.

  See also @timev, @timed, @elapsed, and @allocated.

help?> Int32
search: Int32 UInt32

  Int32 <: Signed

  32-bit signed integer type.
```

Help mode can be exited by pressing backspace at the beginning of the line.

Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as `?` entered help mode when at the beginning of the line, a semicolon (`;`) will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> echo hello
hello
```

Search modes

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an incremental search through the previous history, type `^R` – the control key together with the `r` key. The prompt will change to `(reverse-i-search)`':`, and as you type the search query will appear in the quotes. The most recent result that matches the query will dynamically update to the right of the colon as more is typed. To find an older result using the same query, simply type `^R` again.

Just as `^R` is a reverse search, `^S` is a forward search, with the prompt `(i-search)`':`. The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

84.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (`^D` to exit, `^R` and `^S` for searching), but there are many more. In addition to the control-key, there are also meta-key bindings. These vary more by platform, but most terminals default to using alt- or option- held down with a key to send the meta-key (or can be configured to do so).

Customizing keybindings

Julia's REPL keybindings may be fully customized to a user's preferences by passing a dictionary to `REPL.setup_interface`. The keys of this dictionary may be characters or strings. The key `'*'` refers to the default action. Control plus character `x` bindings are indicated with `"^x"`. Meta plus `x` can be written `"\\Mx"`. The values of the custom keymap must be `nothing` (indicating that the input should be ignored) or functions that accept the signature `(PromptState, AbstractREPL, Char)`. The `REPL.setup_interface` function must be called before the REPL is initialized, by registering the operation with `atreplinit`. For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in `~/.julia/config/startup.jl`:

```
import REPL
import REPL.LineEdit

const mykeys = Dict{Any,Any}(
    # Up Arrow
    "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_prev(s, LineEdit.mode(s).hist)),
    # Down Arrow
    "\e[B" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_next(s, LineEdit.mode(s).hist))
)

function customize_keys(repl)
    repl.interface = REPL.setup_interface(repl; extra_repl_keymap = mykeys)
end

atreplinit(customize_keys)
```

Users should refer to `LineEdit.jl` to discover the available actions on key input.

84.3 Tab completion

In both the Julian and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

```
julia> stri[TAB]
stride     strides     string      strip

julia> Stri[TAB]
StridedArray    StridedMatrix    StridedVecOrMat  StridedVector    String
```

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e₁ = [1,0]
2-element Array{Int64,1}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1×2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2     # √ is equivalent to the sqrt function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)

julia> \h[TAB]
\hat              \hermitconjmatrix  \hkswarow          \hrectangle
\hatapprox        \hexagon           \hookleftarrow     \hrectangleblack
\hbar             \hexagonblack      \hookrightarrow    \hslash
\heartsuit        \hksearow          \house             \hspace

julia> α="\alpha[TAB]"   # LaTeX completion also works in strings
julia> α="α"
```

A full list of tab-completions can be found in the Unicode Input section of the manual.

Completion of paths works for strings and julia's shell mode:

```
julia> path="/[TAB]"
.dockerenv  .juliabox/   boot/        etc/         lib/         media/       opt/         root/        sbin/
↪  sys/         usr/
.dockerinit bin/         dev/         home/        lib64/       mnt/         proc/        run/         srv/
↪  tmp/         var/
shell> /[TAB]
.dockerenv  .juliabox/   boot/        etc/         lib/         media/       opt/         root/        sbin/
↪  sys/         usr/
.dockerinit bin/         dev/         home/        lib64/       mnt/         proc/        run/         srv/
↪  tmp/         var/
```

Tab completion can help with investigation of the available methods matching the input arguments:

```
julia> max([TAB] # All methods are displayed, not shown here due to size of the list

julia> max([1, 2], [TAB] # All methods where `Vector{Int}` matches as first argument
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281

julia> max([1, 2], max(1, 2), [TAB] # All methods matching the arguments.
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281
```

Keywords are also displayed in the suggested methods after `;`, see below line where `limit` and `keepempty` are keyword arguments:

```
julia> split("1 1 1", [TAB]
split(str::AbstractString; limit, keepempty) in Base at strings/util.jl:302
split(str::T, splitter; limit, keepempty) where T<:AbstractString in Base at strings/util.jl:277
```

The completion of the methods uses type inference and can therefore see if the arguments match even if the arguments are output from functions. The function needs to be type stable for the completion to be able to remove non-matching methods.

Tab completion can also help completing fields:

```
julia> import UUIDs

julia> UUIDs.uuid[TAB]
uuid1        uuid4         uuid_version
```

Fields for output from functions can also be completed:

```
julia> split("","")[1].[TAB]
lastindex  offset  string
```

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

Dictionary keys can also be tab completed:

```
julia> foo = Dict("qwer1"=>1, "qwer2"=>2, "asdf"=>3)
Dict{String,Int64} with 3 entries:
  "qwer2" => 2
  "asdf"  => 3
  "qwer1" => 1

julia> foo["q[TAB]

"qwer1" "qwer2"
julia> foo["qwer
```

84.4 Customizing Colors

The colors used by Julia and the REPL can be customized, as well. To change the color of the Julia prompt you can add something like the following to your `~/.julia/config/startup.jl` file, which is to be placed inside your home directory:

```
function customize_colors(repl)
    repl.prompt_color = Base.text_colors[:cyan]
end

atreplinit(customize_colors)
```

The available color keys can be seen by typing `Base.text_colors` in the help mode of the REPL. In addition, the integers 0 to 255 can be used as color keys for terminals with 256 color support.

You can also change the colors for the help and shell prompts and input and answer text by setting the appropriate field of `repl` in the `customize_colors` function above (respectively, `help_color`, `shell_color`, `input_color`, and `answer_color`). For the latter two, be sure that the `envcolors` field is also set to false.

It is also possible to apply boldface formatting by using `Base.text_colors[:bold]` as a color. For instance, to print answers in boldface font, one can use the following as a `~/.julia/config/startup.jl`:

```
function customize_colors(repl)
    repl.envcolors = false
    repl.answer_color = Base.text_colors[:bold]
end

atreplinit(customize_colors)
```

You can also customize the color used to render warning and informational messages by setting the appropriate environment variables. For instance, to render error, warning, and informational messages respectively in magenta, yellow, and cyan you can add the following to your `~/.julia/config/startup.jl` file:

```
ENV["JULIA_ERROR_COLOR"] = :magenta
ENV["JULIA_WARN_COLOR"] = :yellow
ENV["JULIA_INFO_COLOR"] = :cyan
```

Keybinding	Description
Program control	
`^D`	Exit (when buffer is empty)
`^C`	Interrupt or cancel
`^L`	Clear console screen
Return/Enter, `^J`	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
`?` or `;`	Enter help or shell mode (when at start of a line)
`^R`, `^S`	Incremental history search, described above
Cursor movement	
Right arrow, `^F`	Move right one character
Left arrow, `^B`	Move left one character
ctrl-Right, `meta-F`	Move right one word
ctrl-Left, `meta-B`	Move left one word
Home, `^A`	Move to beginning of line
End, `^E`	Move to end of line
Up arrow, `^P`	Move up one line (or change to the previous history entry that matches the text before the cursor)
Down arrow, `^N`	Move down one line (or change to the next history entry that matches the text before the cursor)
Shift-Arrow Key	Move cursor according to the direction of the Arrow key, while activating the region ("shift selection")
Page-up, `meta-P`	Change to the previous history entry
Page-down, `meta-N`	Change to the next history entry
`meta-<`	Change to the first history entry (of the current session if it is before the current position in history)
`meta->`	Change to the last history entry
`^-Space`	Set the "mark" in the editing region (and de-activate the region if it's active)
`^-Space ^-Space`	Set the "mark" in the editing region and make the region "active", i.e. highlighted
`^G`	De-activate the region (i.e. make it not highlighted)
`^X^X`	Exchange the current position with the mark
Editing	
Backspace, `^H`	Delete the previous character, or the whole region when it's active
Delete, `^D`	Forward delete one character (when buffer has text)

Chapter 85

TerminalMenus

TerminalMenus is a submodule of the Julia REPL and enables small, low-profile interactive menus in the terminal.

85.1 Examples

```
import REPL
using REPL.TerminalMenus

options = ["apple", "orange", "grape", "strawberry",
           "blueberry", "peach", "lemon", "lime"]
```

RadioMenu

The RadioMenu allows the user to select one option from the list. The `request` function displays the interactive menu and returns the index of the selected choice. If a user presses 'q' or `ctrl-c`, `request` will return a `-1`.

```
# `pagesize` is the number of items to be displayed at a time.
#  The UI will scroll if the number of options is greater
#   than the `pagesize`
menu = RadioMenu(options, pagesize=4)

# `request` displays the menu and returns the index after the
#   user has selected a choice
choice = request("Choose your favorite fruit:", menu)

if choice != -1
    println("Your favorite fruit is ", options[choice], "!")
else
```

```
    println("Menu canceled.")
end
```

Output:

```
Choose your favorite fruit:
^  grape
   strawberry
 > blueberry
v  peach
Your favorite fruit is blueberry!
```

MultiSelectMenu

The MultiSelectMenu allows users to select many choices from a list.

```
# here we use the default `pagesize` 10
menu = MultiSelectMenu(options)

# `request` returns a `Set` of selected indices
# if the menu us canceled (ctrl-c or q), return an empty set
choices = request("Select the fruits you like:", menu)

if length(choices) > 0
    println("You like the following fruits:")
    for i in choices
        println("  - ", options[i])
    end
else
    println("Menu canceled.")
end
```

Output:

```
Select the fruits you like:
[press: d=done, a=all, n=none]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
```

```
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

85.2 Customization / Configuration

All interface customization is done through the keyword only `TerminalMenus.config()` function.

Arguments

- `charset::Symbol=:na`: ui characters to use (`:ascii` or `:unicode`); overridden by other arguments
- `cursor::Char='>'¦'→'`: character to use for cursor
- `up_arrow::Char='^'¦'↑'`: character to use for up arrow
- `down_arrow::Char='v'¦'↓'`: character to use for down arrow
- `checked::String="[X]"¦"✓"`: string to use for checked
- `unchecked::String="[ ]"¦"⬚")`: string to use for unchecked
- `scroll::Symbol=:na`: If `:wrap` then wrap the cursor around top and bottom, if `:nowrap` do not wrap cursor
- `supress_output::Bool=false`: For testing. If true, menu will not be printed to console.
- `ctrl_c_interrupt::Bool=true`: If `false`, return empty on ^C, if `true` throw InterruptException() on ^C

Examples

```
julia> menu = MultiSelectMenu(options, pagesize=5);


julia> request(menu) # ASCII is used by default
[press: d=done, a=all, n=none]
   [ ] apple
   [X] orange
   [ ] grape
 > [X] strawberry
```

```
v  [ ] blueberry
Set([4, 2])

julia> TerminalMenus.config(charset=:unicode)

julia> request(menu)
[press: d=done, a=all, n=none]
   ⬚ apple
   ✓ orange
   ⬚ grape
 → ✓ strawberry
↓  ⬚ blueberry
Set([4, 2])

julia> TerminalMenus.config(checked="YEP!", unchecked="NOPE", cursor=' ')

julia> request(menu)
[press: d=done, a=all, n=none]
   NOPE apple
   YEP! orange
   NOPE grape
   YEP! strawberry
↓  NOPE blueberry
Set([4, 2])
```

Chapter 86

References

Base.atreplinit – Function.

```
atreplinit(f)
```

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of `f` is the REPL object. This function should be called from within the `.julia/config/startup.jl` initialization file.

source

Chapter 87

Random Numbers

Random number generation in Julia uses the Mersenne Twister library via `MersenneTwister` objects. Julia has a global RNG, which is used by default. Other RNG types can be plugged in by inheriting the `AbstractRNG` type; they can then be used to have multiple streams of random numbers. Besides `MersenneTwister`, Julia also provides the `RandomDevice` RNG type, which is a wrapper over the OS provided entropy.

Most functions related to random generation accept an optional `AbstractRNG` object as first argument, which defaults to the global one if not provided. Moreover, some of them accept optionally dimension specifications `dims...` (which can be given as a tuple) to generate arrays of random values.

A `MersenneTwister` or `RandomDevice` RNG can generate uniformly random numbers of the following types: `Float16`, `Float32`, `Float64`, `BigFloat`, `Bool`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `BigInt` (or complex numbers of those types). Random floating point numbers are generated uniformly in $[0, 1)$. As `BigInt` represents unbounded integers, the interval must be specified (e.g. `rand(big.(1:6))`).

Additionally, normal and exponential distributions are implemented for some `AbstractFloat` and `Complex` types, see `randn` and `randexp` for details.

87.1 Random numbers module

`Random.Random` – Module.

```
Random
```

Support for generating random numbers. Provides `rand`, `randn`, `AbstractRNG`, `MersenneTwister`, and `RandomDevice`.

87.2 Random generation functions

`Base.rand` – Function.

```
rand([rng=GLOBAL_RNG], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by `S`; `S` can be

- an indexable collection (for example `1:9` or `('x', "y", :z)`),
- an `AbstractDict` or `AbstractSet` object,
- a string (considered as a collection of characters), or
- a type: the set of values to pick from is then equivalent to `typemin(S):typemax(S)` for integers (this is not applicable to `BigInt`), to $[0, 1)$ for floating point numbers and to $[0, 1) + i[0, 1)$ for complex floating point numbers;

`S` defaults to `Float64`. When only one argument is passed besides the optional `rng` and is a `Tuple`, it is interpreted as a collection of values (`S`) and not as `dims`.

> Julia 1.1
>
> Support for `S` as a tuple requires at least Julia 1.1.

Examples

```
julia> rand(Int, 2)
2-element Array{Int64,1}:
 1339893410598768192
 1575814717733606317

julia> using Random

julia> rand(MersenneTwister(0), Dict(1=>2, 3=>4))
1=>2

julia> rand((2, 3))
3

julia> rand(Float64, (2, 3))
2×3 Array{Float64,2}:
 0.999717  0.0143835  0.540787
 0.696556  0.783855   0.938235
```

> Note

The complexity of `rand(rng, s::Union{AbstractDict,AbstractSet})` is linear in the length of `s`, unless an optimized method with constant complexity is available, which is the case for `Dict`, `Set` and `BitSet`. For more than a few calls, use `rand(rng, collect(s))` instead, or either `rand(rng, Dict(s))` or `rand(rng, Set(s))` as appropriate.

`Random.rand!` — Function.

```
rand!([rng=GLOBAL_RNG], A, [S=eltype(A)])
```

Populate the array `A` with random values. If `S` is specified (`S` can be a type or a collection, cf. `rand` for details), the values are picked randomly from `S`. This is equivalent to `copyto!(A, rand(rng, S, size(A)))` but without allocating a new array.

Examples

```
julia> rng = MersenneTwister(1234);

julia> rand!(rng, zeros(5))
5-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592
 0.5662374165061859
 0.4600853424625171
 0.7940257103317943
```

`Random.bitrand` — Function.

```
bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a `BitArray` of random boolean values.

Examples

```
julia> rng = MersenneTwister(1234);

julia> bitrand(rng, 10)
10-element BitArray{1}:
 0
 1
 1
 1
```

```
1
0
1
0
0
1
```

`Base.randn` – Function.

```
randn([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type `T` with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default), and their `Complex` counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution of variance 1 (corresponding to real and imaginary part having independent normal distribution with mean zero and variance `1/2`).

Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> randn(rng, ComplexF64)
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, ComplexF32, (2, 3))
2×3 Array{Complex{Float32},2}:
 -0.349649-0.638457im  0.376756-0.192146im  -0.396334-0.0136413im
  0.611224+1.56403im   0.355204-0.365563im  0.0905552+1.31012im
```

`Random.randn!` – Function.

```
randn!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array `A` with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the `rand` function.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randn!(rng, zeros(5))
5-element Array{Float64,1}:
  0.8673472019512456
 -0.9017438158568171
 -0.4944787535042339
 -0.9029142938652416
  0.8644013132535154
```

`Random.randexp` — Function.

```
randexp([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a random number of type `T` according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randexp(rng, Float32)
2.4835055f0

julia> randexp(rng, 3, 3)
3×3 Array{Float64,2}:
 1.5167    1.30652   0.344435
 0.604436  2.78029   0.418516
 0.695867  0.693292  0.643644
```

`Random.randexp!` — Function.

```
randexp!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array `A` with random numbers following the exponential distribution (with scale 1).

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> randexp!(rng, zeros(5))
5-element Array{Float64,1}:
 2.4835053723904896
 1.516703605376473
 0.6044364871025417
 0.6958665886385867
 1.3065196315496677
```

Random.randstring – Function.

```
randstring([rng=GLOBAL_RNG], [chars], [len=8])
```

Create a random string of length `len`, consisting of characters from `chars`, which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> Random.seed!(3); randstring()
"4zSHdXlw"

julia> randstring(MersenneTwister(3), 'a':'z', 6)
"bzlhqn"

julia> randstring("ACGT")
"AGGACATT"
```

> Note
>
> `chars` can be any collection of characters, of type `Char` or `UInt8` (more efficient), provided `rand` can randomly pick characters from it.

87.3 Subsequences, permutations and shuffling

Random.randsubseq – Function.

```
randsubseq([rng=GLOBAL_RNG,] A, p) -> Vector
```

Return a vector consisting of a random subsequence of the given array `A`, where each element of `A` is included (in order) with independent probability `p`. (Complexity is linear in `p*length(A)`, so this function is efficient even if `p` is small and `A` is large.) Technically, this process is known as "Bernoulli sampling" of `A`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randsubseq(rng, 1:8, 0.3)
2-element Array{Int64,1}:
 7
 8
```

Random.randsubseq! – Function.

```
randsubseq!([rng=GLOBAL_RNG,] S, A, p)
```

Like randsubseq, but the results are stored in S (which is resized as needed).

Examples

```
julia> rng = MersenneTwister(1234);

julia> S = Int64[];

julia> randsubseq!(rng, S, 1:8, 0.3)
2-element Array{Int64,1}:
 7
 8

julia> S
2-element Array{Int64,1}:
 7
 8
```

Random.randperm – Function.

```
randperm([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random permutation of length n. The optional rng argument specifies a random number generator (see Random Numbers). The element type of the result is the same as the type of n.

To randomly permute an arbitrary vector, see shuffle or shuffle!.

Julia 1.1

In Julia 1.1 randperm returns a vector v with eltype(v) == typeof(n) while in Julia 1.0 eltype(v) == Int.

Examples

```
julia> randperm(MersenneTwister(1234), 4)
4-element Array{Int64,1}:
 2
 1
 4
 3
```

Random.randperm! – Function.

```
randperm!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in A a random permutation of length length(A). The optional rng argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see shuffle or shuffle!.

Examples

```
julia> randperm!(MersenneTwister(1234), Vector{Int}(undef, 4))
4-element Array{Int64,1}:
 2
 1
 4
 3
```

Random.randcycle – Function.

```
randcycle([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random cyclic permutation of length n. The optional rng argument specifies a random number generator, see Random Numbers. The element type of the result is the same as the type of n.

> Julia 1.1
>
> In Julia 1.1 randcycle returns a vector v with eltype(v) == typeof(n) while in Julia 1.0 eltype(v) == Int.

Examples

```
julia> randcycle(MersenneTwister(1234), 6)
6-element Array{Int64,1}:
 3
```

```
5
4
6
1
2
```

`Random.randcycle!` – Function.

```
randcycle!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in `A` a random cyclic permutation of length `length(A)`. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> randcycle!(MersenneTwister(1234), Vector{Int}(undef, 6))
6-element Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

`Random.shuffle` – Function.

```
shuffle([rng=GLOBAL_RNG,] v::AbstractArray)
```

Return a randomly permuted copy of `v`. The optional `rng` argument specifies a random number generator (see Random Numbers). To permute `v` in-place, see `shuffle!`. To obtain randomly permuted indices, see `randperm`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle(rng, Vector(1:10))
10-element Array{Int64,1}:
  6
  1
 10
  2
```

```
 3
 9
 5
 7
 4
 8
```

`Random.shuffle!` – Function.

```
shuffle!([rng=GLOBAL_RNG,] v::AbstractArray)
```

In-place version of `shuffle`: randomly permute `v` in-place, optionally supplying the random-number generator `rng`.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle!(rng, Vector(1:16))
16-element Array{Int64,1}:
  2
 15
  5
 14
  1
  9
 10
  6
 11
  3
 16
  7
  4
 12
  8
 13
```

87.4 Generators (creation and seeding)

`Random.seed!` – Function.

```
seed!([rng=GLOBAL_RNG], seed) -> rng
seed!([rng=GLOBAL_RNG]) -> rng
```

Reseed the random number generator: `rng` will give a reproducible sequence of numbers if and only if a `seed` is provided. Some RNGs don't accept a seed, like `RandomDevice`. After the call to `seed!`, `rng` is equivalent to a newly created object initialized with the same seed.

If `rng` is not specified, it defaults to seeding the state of the shared thread-local generator.

Examples

```
julia> Random.seed!(1234);

julia> x1 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> Random.seed!(1234);

julia> x2 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> x1 == x2
true

julia> rng = MersenneTwister(1234); rand(rng, 2) == x1
true

julia> MersenneTwister(1) == Random.seed!(rng, 1)
true

julia> rand(Random.seed!(rng), Bool) # not reproducible
true

julia> rand(Random.seed!(rng), Bool)
false
```

```
julia> rand(MersenneTwister(), Bool) # not reproducible either
true
```

Random.AbstractRNG — Type.

```
AbstractRNG
```

Supertype for random number generators such as MersenneTwister and RandomDevice.

Random.MersenneTwister — Type.

```
MersenneTwister(seed)
MersenneTwister()
```

Create a MersenneTwister RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers. The seed may be a non-negative integer or a vector of UInt32 integers. If no seed is provided, a randomly generated one is created (using entropy from the system). See the seed! function for reseeding an already existing MersenneTwister object.

Examples

```
julia> rng = MersenneTwister(1234);

julia> x1 = rand(rng, 2)
2-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592

julia> rng = MersenneTwister(1234);

julia> x2 = rand(rng, 2)
2-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592

julia> x1 == x2
true
```

Random.RandomDevice — Type.

```
RandomDevice()
```

Create a RandomDevice RNG object. Two such objects will always generate different streams of random numbers. The entropy is obtained from the operating system.

87.5 Hooking into the Random API

There are two mostly orthogonal ways to extend Random functionalities:

1. generating random values of custom types
2. creating new generators

The API for 1) is quite functional, but is relatively recent so it may still have to evolve in subsequent releases of the Random module. For example, it's typically sufficient to implement one rand method in order to have all other usual methods work automatically.

The API for 2) is still rudimentary, and may require more work than strictly necessary from the implementor, in order to support usual types of generated values.

Generating random values of custom types

Generating random values for some distributions may involve various trade-offs. Pre-computed values, such as an alias table for discrete distributions, or "squeezing" functions for univariate distributions, can speed up sampling considerably. How much information should be pre-computed can depend on the number of values we plan to draw from a distribution. Also, some random number generators can have certain properties that various algorithms may want to exploit.

The Random module defines a customizable framework for obtaining random values that can address these issues. Each invocation of rand generates a sampler which can be customized with the above trade-offs in mind, by adding methods to Sampler, which in turn can dispatch on the random number generator, the object that characterizes the distribution, and a suggestion for the number of repetitions. Currently, for the latter, Val{1} (for a single sample) and Val{Inf} (for an arbitrary number) are used, with Random.Repetition an alias for both.

The object returned by Sampler is then used to generate the random values. When implementing the random generation interface for a value X that can be sampled from, the implementor should define the method

```
rand(rng, sampler)
```

for the particular sampler returned by Sampler(rng, X, repetition).

Samplers can be arbitrary values that implement rand(rng, sampler), but for most applications the following predefined samplers may be sufficient:

1. `SamplerType{T}()` can be used for implementing samplers that draw from type `T` (e.g. `rand(Int)`). This is the default returned by `Sampler` for types.

2. `SamplerTrivial(self)` is a simple wrapper for `self`, which can be accessed with `[]`. This is the recommended sampler when no pre-computed information is needed (e.g. `rand(1:3)`), and is the default returned by `Sampler` for values.

3. `SamplerSimple(self, data)` also contains the additional `data` field, which can be used to store arbitrary pre-computed values, which should be computed in a custom method of `Sampler`.

We provide examples for each of these. We assume here that the choice of algorithm is independent of the RNG, so we use `AbstractRNG` in our signatures.

`Random.Sampler` — Type.

```
Sampler(rng, x, repetition = Val(Inf))
```

Return a sampler object that can be used to generate random values from `rng` for `x`.

When `sp = Sampler(rng, x, repetition)`, `rand(rng, sp)` will be used to draw random values, and should be defined accordingly.

`repetition` can be `Val(1)` or `Val(Inf)`, and should be used as a suggestion for deciding the amount of precomputation, if applicable.

`Random.SamplerType` and `Random.SamplerTrivial` are default fallbacks for types and values, respectively. `Random.SamplerSimple` can be used to store pre-computed values without defining extra types for only this purpose.

`Random.SamplerType` — Type.

```
SamplerType{T}()
```

A sampler for types, containing no other information. The default fallback for `Sampler` when called with types.

`Random.SamplerTrivial` — Type.

```
SamplerTrivial(x)
```

Create a sampler that just wraps the given value `x`. This is the default fall-back for values. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values without precomputed data.

Random.SamplerSimple – Type.

> SamplerSimple(x, data)
>
> Create a sampler that wraps the given value x and the data. The eltype of this sampler is equal to eltype(x).
>
> The recommended use case is sampling from values with precomputed data.

Decoupling pre-computation from actually generating the values is part of the API, and is also available to the user. As an example, assume that rand(rng, 1:20) has to be called repeatedly in a loop: the way to take advantage of this decoupling is as follows:

```
rng = MersenneTwister()
sp = Random.Sampler(rng, 1:20) # or Random.Sampler(MersenneTwister, 1:20)
for x in X
    n = rand(rng, sp) # similar to n = rand(rng, 1:20)
    # use n
end
```

This is the mechanism that is also used in the standard library, e.g. by the default implementation of random array generation (like in rand(1:20, 10)).

Generating values from a type

Given a type T, it's currently assumed that if rand(T) is defined, an object of type T will be produced. SamplerType is the default sampler for types. In order to define random generation of values of type T, the rand(rng::AbstractRNG, ::Random.SamplerType{T}) method should be defined, and should return values what rand(rng, T) is expected to return.

Let's take the following example: we implement a Die type, with a variable number n of sides, numbered from 1 to n. We want rand(Die) to produce a Die with a random number of up to 20 sides (and at least 4):

```
struct Die
    nsides::Int # number of sides
end

Random.rand(rng::AbstractRNG, ::Random.SamplerType{Die}) = Die(rand(rng, 4:20))

# output
```

Scalar and array methods for Die now work as expected:

```
julia> rand(Die)
Die(18)

julia> rand(MersenneTwister(0), Die)
Die(4)

julia> rand(Die, 3)
3-element Array{Die,1}:
 Die(6)
 Die(11)
 Die(5)

julia> a = Vector{Die}(undef, 3); rand!(a)
3-element Array{Die,1}:
 Die(18)
 Die(6)
 Die(8)
```

A simple sampler without pre-computed data

Here we define a sampler for a collection. If no pre-computed data is required, it can be implemented with a `SamplerTrivial` sampler, which is in fact the default fallback for values.

In order to define random generation out of objects of type `S`, the following method should be defined: `rand(rng::AbstractRNG, sp::Random.SamplerTrivial{S})`. Here, `sp` simply wraps an object of type `S`, which can be accessed via `sp[]`. Continuing the `Die` example, we want now to define `rand(d::Die)` to produce an `Int` corresponding to one of `d`'s sides:

```
julia> Random.rand(rng::AbstractRNG, d::Random.SamplerTrivial{Die}) = rand(rng, 1:d[].nsides);

julia> rand(Die(4))
3

julia> rand(Die(4), 3)
3-element Array{Any,1}:
 3
 4
 2
```

Given a collection type `S`, it's currently assumed that if `rand(::S)` is defined, an object of type `eltype(S)` will be produced. In the last example, a `Vector{Any}` is produced; the reason is that `eltype(Die) == Any`. The remedy is to

define `Base.eltype(::Type{Die}) = Int`.

Generating values for an `AbstractFloat` type

`AbstractFloat` types are special-cased, because by default random values are not produced in the whole type domain, but rather in `[0,1)`. The following method should be implemented for `T <: AbstractFloat`: `Random.rand(::AbstractRNG, ::Random.SamplerTrivial{Random.CloseOpen01{T}})`

An optimized sampler with pre-computed data

Consider a discrete distribution, where numbers `1:n` are drawn with given probabilities that sum to one. When many values are needed from this distribution, the fastest method is using an alias table. We don't provide the algorithm for building such a table here, but suppose it is available in `make_alias_table(probabilities)` instead, and `draw_number(rng, alias_table)` can be used to draw a random number from it.

Suppose that the distribution is described by

```
struct DiscreteDistribution{V <: AbstractVector}
    probabilities::V
end
```

and that we always want to build an a alias table, regardless of the number of values needed (we learn how to customize this below). The methods

```
Random.eltype(::Type{<:DiscreteDistribution}) = Int

function Random.Sampler(::AbstractRng, distribution::DiscreteDistribution, ::Repetition)
    SamplerSimple(disribution, make_alias_table(distribution.probabilities))
end
```

should be defined to return a sampler with pre-computed data, then

```
function rand(rng::AbstractRNG, sp::SamplerSimple{<:DiscreteDistribution})
    draw_number(rng, sp.data)
end
```

will be used to draw the values.

Custom sampler types

The `SamplerSimple` type is sufficient for most use cases with precomputed data. However, in order to demonstrate how to use custom sampler types, here we implement something similar to `SamplerSimple`.

Going back to our `Die` example: `rand(::Die)` uses random generation from a range, so there is an opportunity for this optimization. We call our custom sampler `SamplerDie`.

```
import Random: Sampler, rand

struct SamplerDie <: Sampler{Int} # generates values of type Int
    die::Die
    sp::Sampler{Int} # this is an abstract type, so this could be improved
end

Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerDie(die, Sampler(RNG, 1:die.nsides, r))
# the `r` parameter will be explained later on

rand(rng::AbstractRNG, sp::SamplerDie) = rand(rng, sp.sp)
```

It's now possible to get a sampler with `sp = Sampler(rng, die)`, and use `sp` instead of `die` in any `rand` call involving `rng`. In the simplistic example above, `die` doesn't need to be stored in `SamplerDie` but this is often the case in practice.

Of course, this pattern is so frequent that the helper type used above, namely `Random.SamplerSimple`, is available, saving us the definition of `SamplerDie`: we could have implemented our decoupling with:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerSimple(die, Sampler(RNG, 1:die.nsides, r))

rand(rng::AbstractRNG, sp::SamplerSimple{Die}) = rand(rng, sp.data)
```

Here, `sp.data` refers to the second parameter in the call to the `SamplerSimple` constructor (in this case equal to `Sampler(rng, 1:die.nsides, r)`), while the `Die` object can be accessed via `sp[]`.

Like `SamplerDie`, any custom sampler must be a subtype of `Sampler{T}` where `T` is the type of the generated values. Note that `SamplerSimple(x, data) isa Sampler{eltype(x)}`, so this constrains what the first argument to `SamplerSimple` can be (it's recommended to use `SamplerSimple` like in the `Die` example, where `x` is simply forwarded while defining a `Sampler` method). Similarly, `SamplerTrivial(x) isa Sampler{eltype(x)}`.

Another helper type is currently available for other cases, `Random.SamplerTag`, but is considered as internal API, and can break at any time without proper deprecations.

Using distinct algorithms for scalar or array generation

In some cases, whether one wants to generate only a handful of values or a large number of values will have an impact on the choice of algorithm. This is handled with the third parameter of the `Sampler` constructor. Let's assume we defined two helper types for `Die`, say `SamplerDie1` which should be used to generate only few random values, and `SamplerDieMany` for many values. We can use those types as follows:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{1}) = SamplerDie1(...)
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{Inf}) = SamplerDieMany(...)
```

Of course, `rand` must also be defined on those types (i.e. `rand(::AbstractRNG, ::SamplerDie1)` and `rand(::AbstractRNG, ::SamplerDieMany)`). Note that, as usual, `SamplerTrivial` and `SamplerSimple` can be used if custom types are not necessary.

Note: `Sampler(rng, x)` is simply a shorthand for `Sampler(rng, x, Val(Inf))`, and `Random.Repetition` is an alias for `Union{Val{1}, Val{Inf}}`.

Creating new generators

The API is not clearly defined yet, but as a rule of thumb:

1. any `rand` method producing "basic" types (`isbitstype` integer and floating types in `Base`) should be defined for this specific RNG, if they are needed;
2. other documented `rand` methods accepting an `AbstractRNG` should work out of the box, (provided the methods from 1) what are relied on are implemented), but can of course be specialized for this RNG if there is room for optimization.

Concerning 1), a `rand` method may happen to work automatically, but it's not officially supported and may break without warnings in a subsequent release.

To define a new `rand` method for an hypothetical `MyRNG` generator, and a value specification `s` (e.g. `s == Int`, or `s == 1:10`) of type `S==typeof(s)` or `S==Type{s}` if `s` is a type, the same two methods as we saw before must be defined:

1. `Sampler(::Type{MyRNG}, ::S, ::Repetition)`, which returns an object of type say `SamplerS`
2. `rand(rng::MyRNG, sp::SamplerS)`

It can happen that `Sampler(rng::AbstractRNG, ::S, ::Repetition)` is already defined in the `Random` module. It would then be possible to skip step 1) in practice (if one wants to specialize generation for this particular RNG type), but the corresponding `SamplerS` type is considered as internal detail, and may be changed without warning.

Specializing array generation

In some cases, for a given RNG type, generating an array of random values can be more efficient with a specialized method than by merely using the decoupling technique explained before. This is for example the case for `MersenneTwister`, which natively writes random values in an array.

To implement this specialization for `MyRNG` and for a specification `s`, producing elements of type `S`, the following method can be defined: `rand!(rng::MyRNG, a::AbstractArray{S}, ::SamplerS)`, where `SamplerS` is the type of the sampler returned by `Sampler(MyRNG, s, Val(Inf))`. Instead of `AbstractArray`, it's possible to implement the functionality only for a subtype, e.g. `Array{S}`. The non-mutating array method of `rand` will automatically call this specialization internally.

Chapter 88

SHA

Usage is very straightforward:

```
julia> using SHA

julia> bytes2hex(sha256("test"))
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08"
```

Each exported function (at the time of this writing, SHA-1, SHA-2 224, 256, 384 and 512, and SHA-3 224, 256, 384 and 512 functions are implemented) takes in either an `Array{UInt8}`, a `ByteString` or an `IO` object. This makes it trivial to checksum a file:

```
shell> cat /tmp/test.txt
test
julia> using SHA

julia> open("/tmp/test.txt") do f
           sha2_256(f)
       end
32-element Array{UInt8,1}:
 0x9f
 0x86
 0xd0
 0x81
 0x88
 0x4c
 0x7d
```

```
0x65
   ⋮
0x5d
0x6c
0x15
0xb0
0xf0
0x0a
0x08
```

Note the lack of a newline at the end of `/tmp/text.txt`. Julia automatically inserts a newline before the `julia>` prompt.

Due to the colloquial usage of `sha256` to refer to `sha2_256`, convenience functions are provided, mapping `shaxxx()` function calls to `sha2_xxx()`. For SHA-3, no such colloquialisms exist and the user must use the full `sha3_xxx()` names.

`shaxxx()` takes `AbstractString` and array-like objects (`NTuple` and `Array`) with elements of type `UInt8`.

Note that, at the time of this writing, the SHA3 code is not optimized, and as such is roughly an order of magnitude slower than SHA2.

Chapter 89

Serialization

Serialization.serialize – Function.

```
serialize(stream::IO, value)
```

Write an arbitrary value to a stream in an opaque format, such that it can be read back by deserialize. The read-back value will be as identical as possible to the original. In general, this process will not work if the reading and writing are done by different versions of Julia, or an instance of Julia with a different system image. Ptr values are serialized as all-zero bit patterns (NULL).

An 8-byte identifying header is written to the stream first. To avoid writing the header, construct a Serializer and use it as the first argument to serialize instead. See also Serialization.writeheader.

```
serialize(filename::AbstractString, value)
```

Open a file and serialize the given value to it.

> Julia 1.1
>
> This method is available as of Julia 1.1.

Serialization.deserialize – Function.

```
deserialize(stream)
```

Read a value written by serialize. deserialize assumes the binary data read from stream is correct and has been serialized by a compatible implementation of serialize. It has been designed with simplicity and performance as a goal and does not validate the data read. Malformed data can result in process termination. The caller has to ensure the integrity and correctness of data read from stream.

```
deserialize(filename::AbstractString)
```

Open a file and deserialize its contents.

Julia 1.1

This method is available as of Julia 1.1.

`Serialization.writeheader` – Function.

```
Serialization.writeheader(s::AbstractSerializer)
```

Write an identifying header to the specified serializer. The header consists of 8 bytes as follows:

Offset	Description
0	tag byte (0x37)
1-2	signature bytes "JL"
3	protocol version
4	bits 0-1: endianness: 0 = little, 1 = big
4	bits 2-3: platform: 0 = 32-bit, 1 = 64-bit
5-7	reserved

Chapter 90

Shared Arrays

SharedArrays.SharedArray – Type.

```
SharedArray{T}(dims::NTuple; init=false, pids=Int[])
SharedArray{T,N}(...)
```

Construct a `SharedArray` of a bits type `T` and size `dims` across the processes specified by `pids` - all of which have to be on the same host. If `N` is specified by calling `SharedArray{T,N}(dims)`, then `N` must match the length of `dims`.

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindices` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the `SharedArray` object exists on the node which created the mapping.

```
SharedArray{T}(filename::AbstractString, dims::NTuple, [offset=0]; mode=nothing, init=false, pids=Int[])
SharedArray{T,N}(...)
```

Construct a `SharedArray` backed by the file `filename`, with element type `T` (must be a bits type) and size `dims`, across the processes specified by `pids` - all of which have to be on the same host. This file is mmapped into the host memory, with the following consequences:

- The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)
- Any changes you make to the array values (e.g., `A[3] = 0`) will also change the values on disk

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindices` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

`mode` must be one of `"r"`, `"r+"`, `"w+"`, or `"a+"`, and defaults to `"r+"` if the file specified by `filename` already exists, or `"w+"` if not. If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers. You cannot specify an `init` function if the file is not writable.

`offset` allows you to skip the specified number of bytes at the beginning of the file.

`SharedArrays.SharedVector` — Type.

```
SharedVector
```

A one-dimensional `SharedArray`.

`SharedArrays.SharedMatrix` — Type.

```
SharedMatrix
```

A two-dimensional `SharedArray`.

`Distributed.procs` — Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

`SharedArrays.sdata` — Function.

```
sdata(S::SharedArray)
```

Returns the actual `Array` object backing `S`.

`SharedArrays.indexpids` — Function.

```
indexpids(S::SharedArray)
```

Returns the current worker's index in the list of workers mapping the `SharedArray` (i.e. in the same list returned by `procs(S)`), or 0 if the `SharedArray` is not mapped locally.

`SharedArrays.localindices` — Function.

```
localindices(S::SharedArray)
```

Returns a range describing the "default" indices to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of `1:length(S)`. In multi-process contexts, returns an empty range in the parent process (or any process for which `indexpids` returns 0).

It's worth emphasizing that `localindices` exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a `SharedArray`, all indices should be equally fast for each worker process.

Chapter 91

Sockets

Sockets.Sockets – Module.

Support for sockets. Provides IPAddr and subtypes, TCPSocket, and UDPSocket.

Sockets.connect – Method.

```
connect([host], port::Integer) -> TCPSocket
```

Connect to the host `host` on port `port`.

Sockets.connect – Method.

```
connect(path::AbstractString) -> PipeEndpoint
```

Connect to the named pipe / UNIX domain socket at `path`.

Sockets.listen – Method.

```
listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT) -> TCPServer
```

Listen on port on the address specified by `addr`. By default this listens on `localhost` only. To listen on all interfaces pass `IPv4(0)` or `IPv6(0)` as appropriate. `backlog` determines how many connections can be pending (not having called accept) before the server will begin to reject them. The default value of `backlog` is 511.

Sockets.listen – Method.

```
listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

Sockets.getaddrinfo – Function.

```
getaddrinfo(host::AbstractString, IPAddr=IPv4) -> IPAddr
```

Gets the first IP address of the `host` of the specified `IPAddr` type. Uses the operating system's underlying getaddrinfo implementation, which may do a DNS lookup.

Sockets.getipaddr – Function.

```
getipaddr() -> IPAddr
```

Get an IP address of the local machine, preferring IPv4 over IPv6. Throws if no addresses are available.

```
getipaddr(addr_type::Type{T}) where T<:IPAddr -> T
```

Get an IP address of the local machine of the specified type. Throws if no addresses of the specified type are available.

This function is a backwards-compatibility wrapper around `getipaddrs`. New applications should use `getipaddrs` instead.

Examples

```
julia> getipaddr()
ip"192.168.1.28"

julia> getipaddr(IPv6)
ip"fe80::9731:35af:e1c5:6e49"
```

See also: `getipaddrs`

Sockets.getipaddrs – Function.

```
getipaddrs(addr_type::Type{T}=IPAddr; loopback::Bool=false) where T<:IPAddr -> Vector{T}
```

Get the IP addresses of the local machine.

Setting the optional `addr_type` parameter to `IPv4` or `IPv6` causes only addresses of that type to be returned.

The `loopback` keyword argument dictates whether loopback addresses (e.g. `ip"127.0.0.1"`, `ip"::1"`) are included.

> Julia 1.2
>
> This function is available as of Julia 1.2.

Examples

```
julia> getipaddrs()
5-element Array{IPAddr,1}:
 ip"198.51.100.17"
 ip"203.0.113.2"
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"

julia> getipaddrs(IPv6)
3-element Array{IPv6,1}:
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"
```

See also: islinklocaladdr, `split(ENV["SSH_CONNECTION"], ' ')[3]`

Sockets.getalladdrinfo – Function.

```
getalladdrinfo(host::AbstractString) -> Vector{IPAddr}
```

Gets all of the IP addresses of the `host`. Uses the operating system's underlying `getaddrinfo` implementation, which may do a DNS lookup.

Example

```
julia> getalladdrinfo("google.com")
2-element Array{IPAddr,1}:
 ip"172.217.6.174"
 ip"2607:f8b0:4000:804::200e"
```

Sockets.getnameinfo – Function.

```
getnameinfo(host::IPAddr) -> String
```

Performs a reverse-lookup for IP address to return a hostname and service using the operating system's underlying `getnameinfo` implementation.

Examples

```
julia> getnameinfo(Sockets.IPv4("8.8.8.8"))
"google-public-dns-a.google.com"
```

Sockets.getsockname – Function.

```
getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr, UInt16)
```

Get the IP address and port that the given socket is bound to.

Sockets.getpeername – Function.

```
getpeername(sock::TCPSocket) -> (IPAddr, UInt16)
```

Get the IP address and port of the remote endpoint that the given socket is connected to. Valid only for connected TCP sockets.

Sockets.IPAddr – Type.

```
IPAddr
```

Abstract supertype for IP addresses. IPv4 and IPv6 are subtypes of this.

Sockets.IPv4 – Type.

```
IPv4(host::Integer) -> IPv4
```

Returns an IPv4 object from ip address `host` formatted as an Integer.

Examples

```
julia> IPv4(3223256218)
ip"192.30.252.154"
```

Sockets.IPv6 – Type.

```
IPv6(host::Integer) -> IPv6
```

Returns an IPv6 object from ip address `host` formatted as an Integer.

Examples

```
julia> IPv6(3223256218)
ip"::c01e:fc9a"
```

Sockets.@ip_str – Macro.

```
@ip_str str -> IPAddr
```

Parse `str` as an IP address.

Examples

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> @ip_str "2001:db8:0:0:0:0:2:1"
ip"2001:db8::2:1"
```

Sockets.TCPSocket – Type.

```
TCPSocket(; delay=true)
```

Open a TCP socket using libuv. If `delay` is true, libuv delays creation of the socket's file descriptor till the first `bind` call. `TCPSocket` has various fields to denote the state of the socket as well as its send/receive buffers.

Sockets.UDPSocket – Type.

```
UDPSocket()
```

Open a UDP socket using libuv. `UDPSocket` has various fields to denote the state of the socket.

Sockets.accept – Function.

```
accept(server[, client])
```

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

Sockets.listenany – Function.

```
listenany([host::IPAddr,] port_hint) -> (UInt16, TCPServer)
```

Create a `TCPServer` on any port, using hint as a starting point. Returns a tuple of the actual port that the server was created on and the server itself.

Base.bind – Function.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of `chnl` with a task. `Channel chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed `Channel` objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
ERROR: TaskFailedException:
```

```
foo
Stacktrace:
[...]
```

source

```
bind(socket::Union{UDPSocket, TCPSocket}, host::IPAddr, port::Integer; ipv6only=false, reuseaddr=false, kws...)
```

Bind `socket` to the given `host:port`. Note that `0.0.0.0` will listen on all devices.

- The `ipv6only` parameter disables dual stack mode. If `ipv6only=true`, only an IPv6 stack is created.
- If `reuseaddr=true`, multiple threads or processes can bind to the same address without error if they all set `reuseaddr=true`, but only the last to bind will receive any traffic.

`Sockets.send` – Function.

```
send(socket::UDPSocket, host::IPAddr, port::Integer, msg)
```

Send `msg` over `socket` to `host:port`.

`Sockets.recv` – Function.

```
recv(socket::UDPSocket)
```

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

`Sockets.recvfrom` – Function.

```
recvfrom(socket::UDPSocket) -> (host_port, data)
```

Read a UDP packet from the specified socket, returning a tuple of `(host_port, data)`, where `host_port` will be an InetAddr{IPv4} or InetAddr{IPv6}, as appropriate.

> Julia 1.3
>
> Prior to Julia version 1.3, the first returned value was an address (`IPAddr`). In version 1.3 it was changed to an `InetAddr`.

`Sockets.setopt` – Function.

```
setopt(sock::UDPSocket; multicast_loop=nothing, multicast_ttl=nothing, enable_broadcast=nothing, ttl=nothing)
```

Set UDP socket options.

- `multicast_loop`: loopback for multicast packets (default: `true`).
- `multicast_ttl`: TTL for multicast packets (default: `nothing`).
- `enable_broadcast`: flag must be set to `true` if socket will be used for broadcast messages, or else the UDP system will return an access error (default: `false`).
- `ttl`: Time-to-live of packets sent on the socket (default: `nothing`).

`Sockets.nagle` – Function.

```
nagle(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

Enables or disables Nagle's algorithm on a given TCP server or socket.

`Sockets.quickack` – Function.

```
quickack(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

On Linux systems, the TCP_QUICKACK is disabled or enabled on `socket`.

Chapter 92

Sparse Arrays

Julia has support for sparse vectors and sparse matrices in the `SparseArrays` stdlib module. Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

92.1 Compressed Sparse Column (CSC) Sparse Matrix Storage

In Julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC{Tv,Ti}`, where `Tv` is the type of the stored values, and `Ti` is the integer type for storing column pointers and row indices. The internal representation of `SparseMatrixCSC` is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column i is in colptr[i]:(colptr[i+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of previously unstored entries one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted. If your `SparseMatrixCSC` object contains unsorted row indices, one quick way to sort them is by doing a double transpose.

In some applications, it is convenient to store explicit zero values in a `SparseMatrixCSC`. These are accepted by functions in `Base` (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The `nnz` function returns the number of elements explicitly stored in the sparse data structure, including structural nonzeros. In order to count the exact number of numerical nonzeros, use `count(!iszero, x)`, which inspects every stored element of a sparse matrix. `dropzeros`, and the in-place `dropzeros!`, can be used to remove stored zeros from the sparse matrix.

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [0, 2, 0])
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  0
  [2, 2]  =  2
  [3, 3]  =  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64,Int64} with 1 stored entry:
  [2, 2]  =  2
```

92.2 Sparse Vector Storage

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type `SparseVector{Tv,Ti}` where `Tv` is the type of the stored values and `Ti` the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int              # Length of the sparse vector
    nzind::Vector{Ti}   # Indices of stored values
    nzval::Vector{Tv}   # Stored values, typically nonzeros
end
```

As for `SparseMatrixCSC`, the `SparseVector` type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

92.3 Sparse Vector and Matrix Constructors

The simplest way to create a sparse array is to use a function equivalent to the `zeros` function that Julia provides for working with dense arrays. To produce a sparse array instead, you can use the same name with an `sp` prefix:

```
julia> spzeros(3)
3-element SparseVector{Float64,Int64} with 0 stored entries
```

The `sparse` function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of stored values (this is also known as the COO (coordinate) format). `sparse(I,J,V)` then constructs a sparse matrix such that `S[I[k], J[k]] = V[k]`. The equivalent sparse vector constructor is `sparsevec`, which takes the (row) index vector `I` and the vector `V` with the stored values and constructs a sparse vector `R` such that `R[I[k]] = V[k]`.

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1,  4]  =  1
  [4,  7]  =  2
  [5,  9]  =  3
  [3, 18]  =  -5

julia> R = sparsevec(I,V)
5-element SparseVector{Int64,Int64} with 4 stored entries:
  [1]  =  1
  [3]  =  -5
  [4]  =  2
  [5]  =  3
```

The inverse of the `sparse` and `sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array. `findall(!iszero, x)` returns the cartesian indices of non-zero entries in `x` (including stored entries equal to zero).

```
julia> findnz(S)
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])

julia> findall(!iszero, S)
4-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 4)
 CartesianIndex(4, 7)
 CartesianIndex(5, 9)
 CartesianIndex(3, 18)

julia> findnz(R)
```

```
([1, 3, 4, 5], [1, -5, 2, 3])

julia> findall(!iszero, R)
4-element Array{Int64,1}:
 1
 3
 4
 5
```

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

```
julia> sparse(Matrix(1.0I, 5, 5))
5×5 SparseMatrixCSC{Float64,Int64} with 5 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  1.0
  [5, 5]  =  1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

You can go in the other direction using the `Array` constructor. The `issparse` function can be used to query if a matrix is sparse.

```
julia> issparse(spzeros(5))
true
```

92.4 Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into `(I,J,V)` format using `findnz`, manipulate the values or the structure in the dense vectors `(I,J,V)`, and then reconstruct the sparse matrix.

92.5 Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a given sparse matrix `S`, or that the resulting sparse matrix has density `d`, i.e. each matrix element has a probability `d` of being non-zero.

Details can be found in the Sparse Vectors and Matrices section of the standard library reference.

Sparse	Dense	Description
`spzeros(m,n)`	`zeros(m,n)`	Creates a m-by-n matrix of zeros. (`spzeros(m,n)` is empty.)
`sparse(I, n, n)`	`Matrix(I,n,n)`	Creates a n-by-n identity matrix.
`Array(S)`	`sparse(A)`	Interconverts between dense and sparse formats.
`sprand(m,n,d)`	`rand(m,n)`	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
`sprandn(m,n,d)`	`randn(m,n)`	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
`sprandn(m,n,d,X)`	`randn(m,n,X)`	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed according to the X distribution. (Requires the `Distributions` package.)

Chapter 93

Sparse Arrays

SparseArrays.AbstractSparseArray – Type.

```
AbstractSparseArray{Tv,Ti,N}
```

Supertype for `N`-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. SparseMatrixCSC, SparseVector and `SuiteSparse.CHOLMOD.Sparse` are subtypes of this.

SparseArrays.AbstractSparseVector – Type.

```
AbstractSparseVector{Tv,Ti}
```

Supertype for one-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. Alias for `AbstractSparseArray{Tv,Ti,1}`.

SparseArrays.AbstractSparseMatrix – Type.

```
AbstractSparseMatrix{Tv,Ti}
```

Supertype for two-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. Alias for `AbstractSparseArray{Tv,Ti,2}`.

SparseArrays.SparseVector – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

SparseArrays.SparseMatrixCSC – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format. The standard way of constructing SparseMatrixCSC is through the `sparse` function. See also `spzeros`, `spdiagm` and `sprand`.

`SparseArrays.sparse` – Function.

```
sparse(A)
```

Convert an AbstractMatrix `A` into a sparse matrix.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> sparse(A)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
```

```
sparse(I, J, V,[ m, n, combine])
```

Create a sparse matrix `S` of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of `V` are Booleans in which case `combine` defaults to `|`. All elements of `I` must satisfy `1 <= I[k] <= m`, and all elements of `J` must satisfy `1 <= J[k] <= n`. Numerical zeros in (`I`, `J`, `V`) are retained as structural nonzeros; to drop numerical zeros, use `dropzeros!`.

For additional documentation and an expert driver, see `SparseArrays.sparse!`.

Examples

```
julia> Is = [1; 2; 3];

julia> Js = [1; 2; 3];

julia> Vs = [1; 2; 3];
```

```
julia> sparse(Is, Js, Vs)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  1
  [2, 2]  =  2
  [3, 3]  =  3
```

SparseArrays.sparsevec – Function.

```
sparsevec(I, V, [m, combine])
```

Create a sparse vector `S` of length `m` such that `S[I[k]] = V[k]`. Duplicates are combined using the `combine` function, which defaults to `+` if no `combine` argument is provided, unless the elements of `V` are Booleans in which case `combine` defaults to `|`.

Examples

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];

julia> sparsevec(II, V)
5-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  0.5
  [5]  =  0.2

julia> sparsevec(II, V, 8, -)
8-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  -0.1
  [5]  =  0.2

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false])
3-element SparseVector{Bool,Int64} with 3 stored entries:
  [1]  =  1
  [2]  =  0
  [3]  =  1
```

```
sparsevec(d::Dict, [m])
```

Create a sparse vector of length `m` where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

Examples

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64,Int64} with 2 stored entries:
  [1]  =  3
  [2]  =  2
```

```
sparsevec(A)
```

Convert a vector `A` into a sparse vector of length `m`.

Examples

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  2.0
  [5]  =  3.0
```

SparseArrays.issparse – Function.

```
issparse(S)
```

Returns `true` if `S` is sparse, and `false` otherwise.

Examples

```
julia> sv = sparsevec([1, 4], [2.3, 2.2], 10)
10-element SparseVector{Float64,Int64} with 2 stored entries:
  [1 ]  =  2.3
  [4 ]  =  2.2

julia> issparse(sv)
true

julia> issparse(Array(sv))
false
```

SparseArrays.nnz – Function.

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nnz(A)
3
```

SparseArrays.findnz – Function.

```
findnz(A)
```

Return a tuple `(I, J, V)` where `I` and `J` are the row and column indices of the stored ("structurally non-zero") values in sparse matrix `A`, and `V` is a vector of the values.

Examples

```
julia> A = sparse([1 2 0; 0 0 3; 0 4 0])
3×3 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1, 1]  =  1
  [1, 2]  =  2
  [3, 2]  =  4
  [2, 3]  =  3

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

SparseArrays.spzeros – Function.

```
spzeros([type,]m[,n])
```

Create a sparse vector of length `m` or sparse matrix of size `m x n`. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to `Float64` if not specified.

Examples

```
julia> spzeros(3, 3)
3×3 SparseMatrixCSC{Float64,Int64} with 0 stored entries

julia> spzeros(Float32, 4)
4-element SparseVector{Float32,Int64} with 0 stored entries
```

SparseArrays.spdiagm – Function.

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
spdiagm(m::Integer, n::Ingeger, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a sparse diagonal matrix from `Pair`s of vectors and diagonals. Each vector `kv.second` will be placed on the `kv.first` diagonal. By default (if `size=nothing`), the matrix is square and its size is inferred from `kv`, but a non-square size `m×n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments.

Examples

```
julia> spdiagm(-1 => [1,2,3,4], 1 => [4,3,2,1])
5×5 SparseMatrixCSC{Int64,Int64} with 8 stored entries:
  [2, 1]  =  1
  [1, 2]  =  4
  [3, 2]  =  2
  [2, 3]  =  3
  [4, 3]  =  3
  [3, 4]  =  2
  [5, 4]  =  4
  [4, 5]  =  1
```

SparseArrays.blockdiag – Function.

```
blockdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

```
julia> blockdiag(sparse(2I, 3, 3), sparse(4I, 2, 2))
5×5 SparseMatrixCSC{Int64,Int64} with 5 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2
```

```
  [4, 4]  =  4
  [5, 5]  =  4
```

SparseArrays.sprand – Function.

```
sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

Create a random length `m` sparse vector or `m` by `n` sparse matrix, in which the probability of any element being nonzero is independently given by `p` (and hence the mean density of nonzeros is also exactly `p`). Nonzero values are sampled from the distribution specified by `rfn` and have the type `type`. The uniform distribution is used in case `rfn` is not specified. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> sprand(Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool,Int64} with 1 stored entry:
  [2, 2]  =  1

julia> sprand(Float64, 3, 0.75)
3-element SparseVector{Float64,Int64} with 1 stored entry:
  [3]  =  0.298614
```

SparseArrays.sprandn – Function.

```
sprandn([rng][,Type],m[,n],p::AbstractFloat)
```

Create a random sparse vector of length `m` or sparse matrix of size `m` by `n` with the specified (independent) probability `p` of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional `rng` argument specifies a random number generator, see Random Numbers.

> Julia 1.1
>
> Specifying the output element type `Type` requires at least Julia 1.1.

Examples

```
julia> sprandn(2, 2, 0.75)
2×2 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 2]  =  0.586617
  [2, 2]  =  0.297336
```

SparseArrays.nonzeros – Function.

```
nonzeros(A)
```

Return a vector of the structural nonzero values in sparse array `A`. This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of `A`, and any modifications to the returned vector will mutate `A` as well. See `rowvals` and `nzrange`.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nonzeros(A)
3-element Array{Int64,1}:
 2
 2
 2
```

`SparseArrays.rowvals` — Function.

```
rowvals(A::AbstractSparseMatrixCSC)
```

Return a vector of the row indices of `A`. Any modifications to the returned vector will mutate `A` as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also `nonzeros` and `nzrange`.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> rowvals(A)
3-element Array{Int64,1}:
 1
 2
 3
```

SparseArrays.nzrange – Function.

```
nzrange(A::AbstractSparseMatrixCSC, col::Integer)
```

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with nonzeros and rowvals, this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for j = 1:n
   for i in nzrange(A, j)
      row = rows[i]
      val = vals[i]
      # perform sparse wizardry...
   end
end
```

SparseArrays.droptol! – Function.

```
droptol!(A::AbstractSparseMatrixCSC, tol)
```

Removes stored values from A whose absolute value is less than or equal to tol.

```
droptol!(x::SparseVector, tol)
```

Removes stored values from x whose absolute value is less than or equal to tol.

SparseArrays.dropzeros! – Function.

```
dropzeros!(A::AbstractSparseMatrixCSC;)
```

Removes stored numerical zeros from A.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

```
dropzeros!(x::SparseVector)
```

Removes stored numerical zeros from x.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

SparseArrays.dropzeros – Function.

```
dropzeros(A::AbstractSparseMatrixCSC;)
```

Generates a copy of `A` and removes stored numerical zeros from that copy.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  0.0
  [3, 3]  =  1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  1.0
  [3, 3]  =  1.0
```

```
dropzeros(x::SparseVector)
```

Generates a copy of `x` and removes numerical zeros from that copy.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  0.0
  [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

`SparseArrays.permute` – Function.

```
permute(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
        q::AbstractVector{<:Integer}) where {Tv,Ti}
```

Bilaterally permute `A`, returning `PAQ` (`A[p,q]`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == size(A, 2)`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == size(A, 1)`).

For expert drivers and additional information, see `permute!`.

Examples

```
julia> A = spdiagm(0 => [1, 2, 3, 4], 1 => [5, 6, 7])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [1, 1]  =  1
  [1, 2]  =  5
  [2, 2]  =  2
  [2, 3]  =  6
  [3, 3]  =  3
  [3, 4]  =  7
  [4, 4]  =  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [4, 1]  =  1
  [3, 2]  =  2
  [4, 2]  =  5
  [2, 3]  =  3
  [3, 3]  =  6
  [1, 4]  =  4
  [2, 4]  =  7

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [3, 1]  =  7
  [4, 1]  =  4
  [2, 2]  =  6
  [3, 2]  =  3
  [1, 3]  =  5
  [2, 3]  =  2
  [1, 4]  =  1
```

`Base.permute!` – Method.

```
permute!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
         [C::AbstractSparseMatrixCSC{Tv,Ti}]) where {Tv,Ti}
```

Bilaterally permute `A`, storing result `PAQ` (`A[p,q]`) in `X`. Stores intermediate result `(AQ)^T` (`transpose(A[:,q])`) in optional argument `C` if present. Requires that none of `X`, `A`, and, if present, `C` alias each other; to store result `PAQ` back into `A`, use the following method lacking `X`:

```
permute!(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
         q::AbstractVector{<:Integer}[, C::AbstractSparseMatrixCSC{Tv,Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

`X`'s dimensions must match those of `A` (`size(X, 1) == size(A, 1)` and `size(X, 2) == size(A, 2)`), and `X` must have enough storage to accommodate all allocated entries in `A` (`length(rowvals(X)) >= nnz(A)` and `length(nonzeros(X)) >= nnz(A)`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == size(A, 2)`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == size(A, 1)`).

`C`'s dimensions must match those of `transpose(A)` (`size(C, 1) == size(A, 2)` and `size(C, 2) == size(A, 1)`), and `C` must have enough storage to accommodate all allocated entries in `A` (`length(rowvals(C)) >= nnz(A)` and `length(nonzeros(C)) >= nnz(A)`).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `unchecked_noalias_permute!` and `unchecked_aliasing_permute!`.

See also: `permute`.

Chapter 94

Statistics

The Statistics module contains basic statistics functionality.

`Statistics.std` – Function.

```
std(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample standard deviation of collection `itr`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions, and `means` may contain means for each dimension of `itr`.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> Note
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.stdm` – Function.

```
stdm(itr, mean; corrected::Bool=true)
```

Compute the sample standard deviation of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> Note
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.var` – Function.

```
var(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample variance of collection `itr`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n` is the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> Note
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.varm` – Function.

```
varm(itr, mean; dims, corrected::Bool=true)
```

Compute the sample variance of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating `sum((itr .- mean(itr)).^2) / (length(itr) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> Note
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.cor` – Function.

```
cor(x::AbstractVector)
```

Return the number one.

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix `X` along the dimension `dims`.

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors `x` and `y`.

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices `X` and `Y` along the dimension `dims`.

`Statistics.cov` – Function.

```
cov(x::AbstractVector; corrected::Bool=true)
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

```
cov(X::AbstractMatrix; dims::Int=1, corrected::Bool=true)
```

Compute the covariance matrix of the matrix `X` along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims)`.

```
cov(x::AbstractVector, y::AbstractVector; corrected::Bool=true)
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$ where $*$ denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$.

```
cov(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims::Int=1, corrected::Bool=true)
```

Compute the covariance between the vectors or matrices `X` and `Y` along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims) = size(Y, dims)`.

`Statistics.mean!` – Function.

```
mean!(r, v)
```

Compute the mean of `v` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> using Statistics

julia> v = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Array{Float64,1}:
 1.5
 3.5

julia> mean!([1. 1.], v)
1×2 Array{Float64,2}:
 2.0  3.0
```

`Statistics.mean` – Function.

```
mean(itr)
```

Compute the mean of all elements in a collection.

> Note
>
> If `itr` contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the mean of non-missing values.

Examples

```
julia> using Statistics

julia> mean(1:20)
10.5

julia> mean([1, missing, 3])
missing

julia> mean(skipmissing([1, missing, 3]))
2.0
```

```
mean(f::Function, itr)
```

Apply the function `f` to each element of collection `itr` and take the mean.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

```
mean(f::Function, A::AbstractArray; dims)
```

Apply the function `f` to each element of array `A` and take the mean over dimensions `dims`.

> Julia 1.3
>
> This method requires at least Julia 1.3.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908

julia> mean(√, [1 2 3; 4 5 6], dims=2)
2×1 Array{Float64,2}:
 1.3820881233139908
 2.2285192400943226
```

```
mean(A::AbstractArray; dims)
```

Compute the mean of an array over the given dimensions.

> Julia 1.1
>
> `mean` for empty arrays requires at least Julia 1.1.

Examples

```
julia> using Statistics

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean(A, dims=1)
1×2 Array{Float64,2}:
 2.0  3.0

julia> mean(A, dims=2)
2×1 Array{Float64,2}:
 1.5
 3.5
```

`Statistics.median!` — Function.

```
median!(v)
```

Like median, but may overwrite the input vector.

Statistics.median – Function.

```
median(itr)
```

Compute the median of all elements in a collection. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

> Note
>
> If `itr` contains `NaN` or missing values, the result is also `NaN` or `missing` (`missing` takes precedence if `itr` contains both). Use the skipmissing function to omit `missing` entries and compute the median of non-missing values.

Examples

```
julia> using Statistics

julia> median([1, 2, 3])
2.0

julia> median([1, 2, 3, 4])
2.5

julia> median([1, 2, missing, 4])
missing

julia> median(skipmissing([1, 2, missing, 4]))
2.0
```

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

```
julia> using Statistics

julia> median([1 2; 3 4], dims=1)
1×2 Array{Float64,2}:
 2.0  3.0
```

Statistics.middle – Function.

```
middle(x)
```

Compute the middle of a scalar value, which is equivalent to `x` itself, but of the type of `middle(x, x)` for consistency.

```
middle(x, y)
```

Compute the middle of two reals `x` and `y`, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

```
middle(range)
```

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

```
julia> using Statistics

julia> middle(1:10)
5.5
```

```
middle(a)
```

Compute the middle of an array `a`, which consists of finding its extrema and then computing their mean.

```
julia> using Statistics

julia> a = [1,2,3.6,10.9]
4-element Array{Float64,1}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

Statistics.quantile! – Function.

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false, alpha::Real=1.0, beta::Real=alpha)
```

Compute the quantile(s) of a vector `v` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. If `p` is a vector, an optional output array `q` may also be specified. (If not provided, a new output array is created.) The keyword argument `sorted` indicates whether `v` can be assumed to be sorted; if `false` (the default), then the elements of `v` will be partially sorted in-place.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[k])`, for `k = 1:n` where `n = length(v)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5`
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

  Note

  An `ArgumentError` is thrown if `v` contains `NaN` or `missing` values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", The American Statistician, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> x = [3, 2, 1];

julia> quantile!(x, 0.5)
2.0
```

```
julia> x
3-element Array{Int64,1}:
 1
 2
 3

julia> y = zeros(3);

julia> quantile!(y, x, [0.1, 0.5, 0.9]) === y
true

julia> y
3-element Array{Float64,1}:
 1.2000000000000002
 2.0
 2.8000000000000003
```

Statistics.quantile – Function.

```
quantile(itr, p; sorted=false, alpha::Real=1.0, beta::Real=alpha)
```

Compute the quantile(s) of a collection `itr` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. The keyword argument `sorted` indicates whether `itr` can be assumed to be sorted.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where $x[j]$ is the j-th order statistic, and `γ` is a function of `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `g = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[k])`, for `k = 1:n` where `n = length(itr)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5`
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`

- Def. 9: `alpha=3/8`, `beta=3/8`

  Note

  An `ArgumentError` is thrown if `v` contains `NaN` or `missing` values. Use the `skipmissing` function to omit `missing` entries and compute the quantiles of non-missing values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", The American Statistician, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> quantile(0:20, 0.5)
10.0

julia> quantile(0:20, [0.1, 0.5, 0.9])
3-element Array{Float64,1}:
  2.0
 10.0
 18.000000000000004

julia> quantile(skipmissing([1, 10, missing]), 0.5)
5.5
```

Chapter 95

Unit Testing

95.1 Testing Base Julia

Julia is under rapid development and has an extensive test suite to verify functionality across multiple platforms. If you build Julia from source, you can run this test suite with `make test`. In a binary install, you can run the test suite using `Base.runtests()`.

`Base.runtests` – Function.

```
Base.runtests(tests=["all"]; ncores=ceil(Int, Sys.CPU_THREADS / 2),
              exit_on_error=false, revise=false, [seed])
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `ncores` processors. If `exit_on_error` is `false`, when one test fails, all remaining tests in other files will still be run; they are otherwise discarded, when `exit_on_error == true`. If `revise` is `true`, the `Revise` package is used to load any modifications to `Base` or to the standard libraries before running the tests. If a seed is provided via the keyword argument, it is used to seed the global RNG in the context where the tests are run; otherwise the seed is chosen randomly.

source

95.2 Basic Unit Tests

The `Test` module provides simple unit testing functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete.

Simple unit testing can be performed with the `@test` and `@test_throws` macros:

`Test.@test` – Macro.

```
@test ex
@test f(args...) key=val ...
```

Tests that the expression `ex` evaluates to `true`. Returns a `Pass` `Result` if it does, a `Fail` `Result` if it is `false`, and an `Error` `Result` if it could not be evaluated.

Examples

```
julia> @test true
Test Passed


julia> @test [1, 2] + [2, 1] == [3, 3]
Test Passed
```

The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
julia> @test π ≈ 3.14 atol=0.01
Test Passed
```

This is equivalent to the uglier test `@test ≈(π, 3.14, atol=0.01)`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments (`k=v`).

`Test.@test_throws` – Macro.

```
@test_throws exception expr
```

Tests that the expression `expr` throws `exception`. The exception may specify either a type, or a value (which will be tested for equality by comparing fields). Note that `@test_throws` does not support a trailing keyword form.

Examples

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
      Thrown: BoundsError


julia> @test_throws DimensionMismatch [1, 2, 3] + [1, 2]
Test Passed
      Thrown: DimensionMismatch
```

For example, suppose we want to check our new function `foo(x)` works as expected:

```
julia> using Test

julia> foo(x) = length(x)^2
foo (generic function with 1 method)
```

If the condition is true, a `Pass` is returned:

```
julia> @test foo("bar") == 9
Test Passed

julia> @test foo("fizz") >= 10
Test Passed
```

If the condition is false, then a `Fail` is returned and an exception is thrown:

```
julia> @test foo("f") == 20
Test Failed at none:1
  Expression: foo("f") == 20
   Evaluated: 1 == 20
ERROR: There was an error during testing
```

If the condition could not be evaluated because an exception was thrown, which occurs in this case because `length` is not defined for symbols, an `Error` object is returned and an exception is thrown:

```
julia> @test foo(:cat) == 1
Error During Test
  Test threw an exception of type MethodError
  Expression: foo(:cat) == 1
  MethodError: no method matching length(::Symbol)
  Closest candidates are:
    length(::SimpleVector) at essentials.jl:256
    length(::Base.MethodList) at reflection.jl:521
    length(::MethodTable) at reflection.jl:597
    ...
  Stacktrace:
  [...]
ERROR: There was an error during testing
```

If we expect that evaluating an expression should throw an exception, then we can use `@test_throws` to check that this occurs:

```
julia> @test_throws MethodError foo(:cat)
Test Passed
      Thrown: MethodError
```

95.3 Working with Test Sets

Typically a large number of tests are used to make sure functions work correctly over a range of inputs. In the event a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

The `@testset` macro can be used to group tests into sets. All the tests in a test set will be run, and at the end of the test set a summary will be printed. If any of the tests failed, or could not be evaluated due to an error, the test set will then throw a `TestSetException`.

`Test.@testset` – Macro.

```
@testset [CustomTestSet] [option=val  ...] ["description"] begin ... end
@testset [CustomTestSet] [option=val  ...] ["description $v"] for v in (...) ... end
@testset [CustomTestSet] [option=val  ...] ["description $v, $w"] for v in (...), w in (...) ... end
```

Starts a new test set, or multiple test sets if a `for` loop is provided.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fail`s or `Error`s, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of `AbstractTestSet`) can be given and it will also be used for any nested `@testset` invocations. The given options are only applied to the test set where they are given. The default test set type does not take any options.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables.

By default the `@testset` macro will return the testset object itself, though this behavior can be customized in other testset types. If a `for` loop is used then the macro collects and returns a list of the return values of the `finish` method, which by default will return a list of the testset objects used in each iteration.

Before the execution of the body of a `@testset`, there is an implicit call to `Random.seed!(seed)` where `seed` is the current seed of the global RNG. Moreover, after the execution of the body, the state of the global RNG is restored

to what it was before the `@testset`. This is meant to ease reproducibility in case of failure, and to allow seamless re-arrangements of `@testset`s regardless of their side-effect on the global RNG state.

Examples

```
julia> @testset "trigonometric identities" begin
           θ = 2/3*π
           @test sin(-θ) ≈ -sin(θ)
           @test cos(-θ) ≈ cos(θ)
           @test sin(2θ) ≈ 2*sin(θ)*cos(θ)
           @test cos(2θ) ≈ cos(θ)^2 - sin(θ)^2
       end;
Test Summary:            | Pass  Total
trigonometric identities |    4      4
```

We can put our tests for the `foo(x)` function in a test set:

```
julia> @testset "Foo Tests" begin
           @test foo("a")   == 1
           @test foo("ab")  == 4
           @test foo("abc") == 9
       end;
Test Summary: | Pass  Total
Foo Tests     |    3      3
```

Test sets can also be nested:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(fill(1.0, i)) == i^2
           end
       end;
Test Summary: | Pass  Total
Foo Tests     |    8      8
```

In the event that a nested test set has no failures, as happened here, it will be hidden in the summary. If we do have a test failure, only the details for the failed test sets will be shown:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @testset "Felines" begin
                   @test foo("cat") == 9
               end
               @testset "Canines" begin
                   @test foo("dog") == 9
               end
           end
           @testset "Arrays" begin
               @test foo(zeros(2)) == 4
               @test foo(fill(1.0, 4)) == 15
           end
       end

Arrays: Test Failed
  Expression: foo(fill(1.0, 4)) == 15
   Evaluated: 16 == 15
[...]
Test Summary: | Pass  Fail  Total
Foo Tests     |    3     1      4
  Animals     |    2            2
  Arrays      |    1     1      2
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

95.4 Other Test Macros

As calculations on floating-point values can be imprecise, you can perform approximate equality checks using either `@test a ≈ b` (where ≈, typed via tab completion of `\approx`, is the `isapprox` function) or use `isapprox` directly.

```
julia> @test 1 ≈ 0.999999999
Test Passed

julia> @test 1 ≈ 0.999999
Test Failed at none:1
  Expression: 1 ≈ 0.999999
```

```
   Evaluated: 1 ≈ 0.999999
ERROR: There was an error during testing
```

`Test.@inferred` – Macro.

```
@inferred [AllowedType] f(x)
```

Tests that the call expression `f(x)` returns a value of the same type inferred by the compiler. It is useful to check for type stability.

`f(x)` can be any call expression. Returns the result of `f(x)` if the types match, and an `Error Result` if it finds different types.

Optionally, `AllowedType` relaxes the test, by making it pass when either the type of `f(x)` matches the inferred type modulo `AllowedType`, or when the return type is a subtype of `AllowedType`. This is useful when testing type stability of functions returning a small union such as `Union{Nothing, T}` or `Union{Missing, T}`.

```
julia> f(a) = a > 1 ? 1 : 1.0
f (generic function with 1 method)

julia> typeof(f(2))
Int64

julia> @code_warntype f(2)
Variables
  #self#::Core.Compiler.Const(f, false)
  a::Int64

Body::UNION{FLOAT64, INT64}
1 — %1 = (a > 1)::Bool
└──      goto #3 if not %1
2 —      return 1
3 —      return 1.0

julia> @inferred f(2)
ERROR: return type Int64 does not match inferred return type Union{Float64, Int64}
[...]

julia> @inferred max(1, 2)
2
```

```
julia> g(a) = a < 10 ? missing : 1.0
g (generic function with 1 method)

julia> @inferred g(20)
ERROR: return type Float64 does not match inferred return type Union{Missing, Float64}
[...]

julia> @inferred Missing g(20)
1.0

julia> h(a) = a < 10 ? missing : f(a)
h (generic function with 1 method)

julia> @inferred Missing h(20)
ERROR: return type Int64 does not match inferred return type Union{Missing, Float64, Int64}
[...]
```

`Test.@test_logs` – Macro.

```
@test_logs [log_patterns...] [keywords] expression
```

Collect a list of log records generated by `expression` using `collect_test_logs`, check that they match the sequence `log_patterns`, and return the value of `expression`. The `keywords` provide some simple filtering of log records: the `min_level` keyword controls the minimum log level which will be collected for the test, the `match_mode` keyword defines how matching will be performed (the default `:all` checks that all logs and patterns match pairwise; use `:any` to check that the pattern matches at least once somewhere in the sequence.)

The most useful log pattern is a simple tuple of the form `(level,message)`. A different number of tuple elements may be used to match other log metadata, corresponding to the arguments to passed to `AbstractLogger` via the `handle_message` function: `(level,message,module,group,id,file,line)`. Elements which are present will be matched pairwise with the log record fields using `==` by default, with the special cases that `Symbol`s may be used for the standard log levels, and `Regex`s in the pattern will match string or Symbol fields using `occursin`.

Examples

Consider a function which logs a warning, and several debug messages:

```
function foo(n)
    @info "Doing foo with n=$n"
    for i=1:n
        @debug "Iteration $i"
```

```
    end
    42
end
```

We can test the info message using

```
@test_logs (:info,"Doing foo with n=2") foo(2)
```

If we also wanted to test the debug messages, these need to be enabled with the `min_level` keyword:

```
@test_logs (:info,"Doing foo with n=2") (:debug,"Iteration 1") (:debug,"Iteration 2") min_level=Debug foo(2)
```

If you want to test that some particular messages are generated while ignoring the rest, you can set the keyword `match_mode=:any`:

```
@test_logs (:info,) (:debug,"Iteration 42") min_level=Debug match_mode=:any foo(100)
```

The macro may be chained with `@test` to also test the returned value:

```
@test (@test_logs (:info,"Doing foo with n=2") foo(2)) == 42
```

`Test.@test_deprecated` – Macro.

```
@test_deprecated [pattern] expression
```

When `--depwarn=yes`, test that `expression` emits a deprecation warning and return the value of `expression`. The log message string will be matched against `pattern` which defaults to `r"deprecated"i`.

When `--depwarn=no`, simply return the result of executing `expression`. When `--depwarn=error`, check that an ErrorException is thrown.

Examples

```
# Deprecated in julia 0.7
@test_deprecated num2hex(1)

# The returned value can be tested by chaining with @test:
@test (@test_deprecated num2hex(1)) == "0000000000000001"
```

`Test.@test_warn` – Macro.

```
@test_warn msg expr
```

Test whether evaluating `expr` results in `stderr` output that contains the `msg` string or matches the `msg` regular expression. If `msg` is a boolean function, tests whether `msg(output)` returns `true`. If `msg` is a tuple or array, checks that the error output contains/matches each item in `msg`. Returns the result of evaluating `expr`.

See also `@test_nowarn` to check for the absence of error output.

Note: Warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs` instead.

`Test.@test_nowarn` – Macro.

```
@test_nowarn expr
```

Test whether evaluating `expr` results in empty `stderr` output (no warnings or other messages). Returns the result of evaluating `expr`.

Note: The absence of warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs expr` instead.

95.5 Broken Tests

If a test fails consistently it can be changed to use the `@test_broken` macro. This will denote the test as `Broken` if the test continues to fail and alerts the user via an `Error` if the test succeeds.

`Test.@test_broken` – Macro.

```
@test_broken ex
@test_broken f(args...) key=val ...
```

Indicates a test that should pass but currently consistently fails. Tests that the expression `ex` evaluates to `false` or causes an exception. Returns a `Broken Result` if it does, or an `Error Result` if the expression evaluates to `true`.

The `@test_broken f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_broken 1 == 2
Test Broken
  Expression: 1 == 2


julia> @test_broken 1 == 2 atol=0.1
Test Broken
  Expression: ==(1, 2, atol = 0.1)
```

`@test_skip` is also available to skip a test without evaluation, but counting the skipped test in the test set reporting. The test will not run but gives a `Broken Result`.

`Test.@test_skip` – Macro.

```
@test_skip ex
@test_skip f(args...) key=val ...
```

Marks a test that should not be executed but should be included in test summary reporting as `Broken`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality.

The `@test_skip f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_skip 1 == 2
Test Broken
  Skipped: 1 == 2

julia> @test_skip 1 == 2 atol=0.1
Test Broken
  Skipped: ==(1, 2, atol = 0.1)
```

95.6 Creating Custom `AbstractTestSet` Types

Packages can create their own `AbstractTestSet` subtypes by implementing the `record` and `finish` methods. The subtype should have a one-argument constructor taking a description string, with any options passed in as keyword arguments.

`Test.record` – Function.

```
record(ts::AbstractTestSet, res::Result)
```

Record a result to a testset. This function is called by the `@testset` infrastructure each time a contained `@test` macro completes, and is given the test result (which could be an `Error`). This will also be called with an `Error` if an exception is thrown inside the test block but outside of a `@test` context.

`Test.finish` – Function.

```
finish(ts::AbstractTestSet)
```

Do any final processing necessary for the given testset. This is called by the `@testset` infrastructure after a test block executes. One common use for this function is to record the testset to the parent's results list, using `get_testset`.

`Test` takes responsibility for maintaining a stack of nested testsets as they are executed, but any result accumulation is the responsibility of the `AbstractTestSet` subtype. You can access this stack with the `get_testset` and `get_testset_depth` methods. Note that these functions are not exported.

`Test.get_testset` – Function.

```
get_testset()
```

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

`Test.get_testset_depth` – Function.

```
get_testset_depth()
```

Returns the number of active test sets, not including the default test set

`Test` also makes sure that nested `@testset` invocations use the same `AbstractTestSet` subtype as their parent unless it is set explicitly. It does not propagate any properties of the testset. Option inheritance behavior can be implemented by packages using the stack infrastructure that `Test` provides.

Defining a basic `AbstractTestSet` subtype might look like:

```
import Test: record, finish
using Test: AbstractTestSet, Result, Pass, Fail, Error
using Test: get_testset_depth, get_testset
struct CustomTestSet <: Test.AbstractTestSet
    description::AbstractString
    foo::Int
    results::Vector
    # constructor takes a description string and options keyword arguments
    CustomTestSet(desc; foo=1) = new(desc, foo, [])
end

record(ts::CustomTestSet, child::AbstractTestSet) = push!(ts.results, child)
record(ts::CustomTestSet, res::Result) = push!(ts.results, res)
function finish(ts::CustomTestSet)
    # just record if we're not the top-level parent
```

```
    if get_testset_depth() > 0
        record(get_testset(), ts)
    end
    ts
end
```

And using that testset looks like:

```
@testset CustomTestSet foo=4 "custom testset inner 2" begin
    # this testset should inherit the type, but not the argument.
    @testset "custom testset inner" begin
        @test true
    end
end
```

Chapter 96

UUIDs

UUIDs.uuid1 – Function.

```
uuid1([rng::AbstractRNG=GLOBAL_RNG]) -> UUID
```

Generates a version 1 (time-based) universally unique identifier (UUID), as specified by RFC 4122. Note that the Node ID is randomly generated (does not identify the host) according to section 4.5 of the RFC.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid1(rng)
UUID("cfc395e8-590f-11e8-1f13-43a2532b2fa8")
```

UUIDs.uuid4 – Function.

```
uuid4([rng::AbstractRNG=GLOBAL_RNG]) -> UUID
```

Generates a version 4 (random or pseudo-random) universally unique identifier (UUID), as specified by RFC 4122.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid4(rng)
UUID("196f2941-2d58-45ba-9f13-43a2532b2fa8")
```

UUIDs.uuid5 – Function.

```
uuid5(ns::UUID, name::String) -> UUID
```

Generates a version 5 (namespace and domain-based) universally unique identifier (UUID), as specified by RFC 4122.

> Julia 1.1
>
> This function requires at least Julia 1.1.

Examples

```
julia> rng = MersenneTwister(1234);

julia> u4 = uuid4(rng)
UUID("196f2941-2d58-45ba-9f13-43a2532b2fa8")

julia> u5 = uuid5(u4, "julia")
UUID("b37756f8-b0c0-54cd-a466-19b3d25683bc")
```

`UUIDs.uuid_version` – Function.

```
uuid_version(u::UUID) -> Int
```

Inspects the given UUID and returns its version (see RFC 4122).

Examples

```
julia> uuid_version(uuid4())
4
```

Chapter 97

Unicode

`Unicode.isassigned` – Function.

```
Unicode.isassigned(c) -> Bool
```

Returns `true` if the given char or integer is an assigned Unicode code point.

Examples

```
julia> Unicode.isassigned(101)
true

julia> Unicode.isassigned('\x01')
true
```

`Unicode.normalize` – Function.

```
Unicode.normalize(s::AbstractString; keywords...)
Unicode.normalize(s::AbstractString, normalform::Symbol)
```

Normalize the string `s`. By default, canonical composition (`compose=true`) is performed without ensuring Unicode versioning stability (`compat=false`), which produces the shortest possible equivalent string but may introduce composition characters not present in earlier Unicode versions.

Alternatively, one of the four "normal forms" of the Unicode standard can be specified: `normalform` can be `:NFC`, `:NFD`, `:NFKC`, or `:NFKD`. Normal forms C (canonical composition) and D (canonical decomposition) convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize "compatibility equivalents": they

convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be obtained by calling `Unicode.normalize(s; keywords...)`, where any number of the following boolean keywords options (which all default to `false` except for `compose`) are specified:

- `compose=false`: do not perform canonical composition
- `decompose=true`: do canonical decomposition instead of canonical composition (`compose=true` is ignored if present)
- `compat=true`: compatibility equivalents are canonicalized
- `casefold=true`: perform Unicode case folding, e.g. for case-insensitive string comparison
- `newline2lf=true`, `newline2ls=true`, or `newline2ps=true`: convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- `stripmark=true`: strip diacritical marks (e.g. accents)
- `stripignore=true`: strip Unicode's "default ignorable" characters (e.g. the soft hyphen or the left-to-right marker)
- `stripcc=true`: strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified
- `rejectna=true`: throw an error if unassigned code points are found
- `stable=true`: enforce Unicode versioning stability (never introduce characters missing from earlier Unicode versions)

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

Examples

```
julia> "é" == Unicode.normalize("e´") #LHS: Unicode U+00e9, RHS: U+0065 & U+0301
true

julia> "μ" == Unicode.normalize("μ", compat=true) #LHS: Unicode U+03bc, RHS: Unicode U+00b5
true

julia> Unicode.normalize("JuLiA", casefold=true)
"julia"

julia> Unicode.normalize("JúLiA", stripmark=true)
"JuLiA"
```

`Unicode.graphemes` — Function.

```
graphemes(s::AbstractString) -> GraphemeIterator
```

Returns an iterator over substrings of `s` that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

Part IV

개발자 문서

Chapter 98

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

98.1 Module bindings

The exported names for a `Module` are available using `names(m::Module)`, which will return an array of `Symbol` elements representing the exported bindings. `names(m::Module, all = true)` returns symbols for all bindings in `m`, regardless of export status.

98.2 DataType fields

The names of `DataType` fields may be interrogated using `fieldnames`. For example, given the following type, `fieldnames(Point)` returns a tuple of `Symbol`s representing the field names:

```
julia> struct Point
           x::Int
           y
       end

julia> fieldnames(Point)
(:x, :y)
```

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

```
julia> Point.types
svec(Int64, Any)
```

While `x` is annotated as an `Int`, `y` was unannotated in the type definition, therefore `y` defaults to the `Any` type.

Types are themselves represented as a structure called `DataType`:

```
julia> typeof(Point)
DataType
```

Note that `fieldnames(DataType)` gives the names for each field of `DataType` itself, and one of these fields is the `types` field observed in the example above.

98.3 Subtypes

The direct subtypes of any `DataType` may be listed using `subtypes`. For example, the abstract `DataType` `AbstractFloat` has four (concrete) subtypes:

```
julia> subtypes(AbstractFloat)
4-element Array{Any,1}:
 BigFloat
 Float16
 Float32
 Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of `subtypes` may be used to inspect the full type tree.

98.4 DataType layout

The internal representation of a `DataType` is critically important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if `T` is stored with C-compatible alignment. `fieldoffset(T::DataType, i::Integer)` returns the (byte) offset for field i relative to the start of the type.

98.5 Function methods

The methods of any generic function may be listed using `methods`. The method dispatch table may be searched for methods accepting a given type using `methodswith`.

98.6 Expansion and lowering

As discussed in the Metaprogramming section, the `macroexpand` function gives the unquoted and interpolated expression (`Expr`) form for a given macro. To use `macroexpand`, `quote` the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand(@__MODULE__, :(@edit println("")) )
:(InteractiveUtils.edit(println, (Base.typesof)("")))
```

The functions `Base.Meta.show_sexpr` and `dump` are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the `Meta.lower` function gives the `lowered` form of any expression and is of particular interest for understanding how language constructs map to primitive operations such as assignments, branches, and calls:

```
julia> Meta.lower(@__MODULE__, :( [1+2, sin(0.5)] ))
:($(Expr(:thunk, CodeInfo(
    @ none within `top-level scope'
1 — %1 = 1 + 2
│   %2 = sin(0.5)
│   %3 = Base.vect(%1, %2)
└──      return %3
))))
```

98.7 Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using `code_lowered`, and the type-inferred form is available using `code_typed`. `code_warntype` adds highlighting to the output of `code_typed`.

Closer to the machine, the LLVM intermediate representation of a function may be printed using by `code_llvm`, and finally the compiled machine code is available using `code_native` (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

```
julia> @code_llvm +(1,1)
```

```
define i64 @"julia_+_130862"(i64, i64) {
top:
    %2 = add i64 %1, %0
    ret i64 %2
}
```

For more informations see `@code_lowered`, `@code_typed`, `@code_warntype`, `@code_llvm`, and `@code_native`.

Printing of debug information

The aforementioned functions and macros take the keyword argument `debuginfo` that controls the level debug information printed.

```
julia> @code_typed debuginfo=:source +(1,1)
CodeInfo(
    @ int.jl:53 within `+'
1 ─ %1 = Base.add_int(x, y)::Int64└──
      return %1
) => Int64
```

Possible values for `debuginfo` are: `:none`, `:source`, and`:default`. Per default debug information is not printed, but that can be changed by setting `Base.IRShow.default_debuginfo[] = :source`.

Chapter 99

Documentation of Julia's Internals

99.1 Initialization of the Julia runtime

How does the Julia runtime execute `julia -e 'println("Hello World!")'` ?

`main()`

Execution starts at `main()` in `ui/repl.c`.

`main()` calls `libsupport_init()` to set the C library locale and to initialize the "ios" library (see `ios_init_stdstreams()` and Legacy `ios.c` library).

Next `jl_parse_opts()` is called to process command line options. Note that `jl_parse_opts()` only deals with options that affect code generation or early initialization. Other options are handled later by `process_options()` in `base/client.jl`.

`jl_parse_opts()` stores command line options in the global `jl_options` struct.

`julia_init()`

`julia_init()` in `task.c` is called by `main()` and calls `_julia_init()` in `init.c`.

`_julia_init()` begins by calling `libsupport_init()` again (it does nothing the second time).

`restore_signals()` is called to zero the signal handler mask.

`jl_resolve_sysimg_location()` searches configured paths for the base system image. See Building the Julia system image.

`jl_gc_init()` sets up allocation pools and lists for weak refs, preserved values and finalization.

`jl_init_frontend()` loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

`jl_init_types()` creates `jl_datatype_t` type description objects for the built-in types defined in `julia.h`. e.g.

```
jl_any_type = jl_new_abstracttype(jl_symbol("Any"), core, NULL, jl_emptysvec);
jl_any_type->super = jl_any_type;

jl_type_type = jl_new_abstracttype(jl_symbol("Type"), core, jl_any_type, jl_emptysvec);

jl_int32_type = jl_new_primitivetype(jl_symbol("Int32"), core,
                                     jl_any_type, jl_emptysvec, 32);
```

`jl_init_tasks()` creates the `jl_datatype_t* jl_task_type` object; initializes the global `jl_root_task` struct; and sets `jl_current_task` to the root task.

`jl_init_codegen()` initializes the LLVM library.

`jl_init_serializer()` initializes 8-bit serialization tags for builtin `jl_value_t` values.

If there is no sysimg file (`!jl_options.image_file`) then the `Core` and `Main` modules are created and `boot.jl` is evaluated:

`jl_core_module = jl_new_module(jl_symbol("Core"))` creates the Julia `Core` module.

`jl_init_intrinsic_functions()` creates a new Julia module `Intrinsics` containing constant `jl_intrinsic_type` symbols. These define an integer code for each intrinsic function. `emit_intrinsic()` translates these symbols into LLVM instructions during code generation.

`jl_init_primitives()` hooks C functions up to Julia function symbols. e.g. the symbol `Core.:(===)()` is bound to C function pointer `jl_f_is()` by calling `add_builtin_func("===", jl_f_is)`.

`jl_new_main_module()` creates the global "Main" module and sets `jl_current_task->current_module = jl_main_module`.

Note: `_julia_init()` then sets `jl_root_task->current_module = jl_core_module`. `jl_root_task` is an alias of `jl_current_task` at this point, so the `current_module` set by `jl_new_main_module()` above is overwritten.

`jl_load("boot.jl", sizeof("boot.jl"))` calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute `boot.jl`. <!– TODO – drill down into eval? –>

`jl_get_builtin_hooks()` initializes global C pointers to Julia globals defined in `boot.jl`.

`jl_init_box_caches()` pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
jl_value_t *jl_box_uint8(uint32_t x)
{
    return boxed_uint8_cache[(uint8_t)x];
}
```

`_julia_init()` iterates over the `jl_core_module->bindings.table` looking for `jl_datatype_t` values and sets the type name's module prefix to `jl_core_module`.

`jl_add_standard_imports(jl_main_module)` does "using Base" in the "Main" module.

Note: `_julia_init()` now reverts to `jl_root_task->current_module = jl_main_module` as it was before being set to `jl_core_module` above.

Platform specific signal handlers are initialized for `SIGSEGV` (OSX, Linux), and `SIGFPE` (Windows).

Other signals (`SIGINFO, SIGBUS, SIGILL, SIGTERM, SIGABRT, SIGQUIT, SIGSYS` and `SIGPIPE`) are hooked up to `sigdie_handler()` which prints a backtrace.

`jl_init_restored_modules()` calls `jl_module_run_initializer()` for each deserialized module to run the `__init__()` function.

Finally `sigint_handler()` is hooked up to `SIGINT` and calls `jl_throw(jl_interrupt_exception)`.

`_julia_init()` then returns back to `main()` in `ui/repl.c` and `main()` calls `true_main(argc, (char**)argv)`.

> sysimg
>
> If there is a sysimg file, it contains a pre-cooked image of the `Core` and `Main` modules (and whatever else is created by `boot.jl`). See Building the Julia system image.
>
> `jl_restore_system_image()` deserializes the saved sysimg into the current Julia runtime environment and initialization continues after `jl_init_box_caches()` below...
>
> Note: `jl_restore_system_image()` (and `staticdata.c` in general) uses the Legacy `ios.c` library.

`true_main()`

`true_main()` loads the contents of `argv[]` into `Base.ARGS`.

If a `.jl` "program" file was supplied on the command line, then `exec_program()` calls `jl_load(program,len)` which calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute the program.

However, in our example (`julia -e 'println("Hello World!")'`), `jl_get_global(jl_base_module, jl_symbol("_start"))` looks up `Base._start` and `jl_apply()` executes it.

`Base._start`

`Base._start` calls `Base.process_options` which calls `jl_parse_input_line("println("Hello World!")")` to create an expression object and `Base.eval()` to execute it.

Base.eval

`Base.eval()` was mapped to `jl_f_top_eval` by `jl_init_primitives()`.

`jl_f_top_eval()` calls `jl_toplevel_eval_in(jl_main_module, ex)`, where `ex` is the parsed expression `println("Hello World!")`.

`jl_toplevel_eval_in()` calls `jl_toplevel_eval_flex()` which calls `eval()` in `interpreter.c`.

The stack dump below shows how the interpreter works its way through various methods of `Base.println()` and `Base.print()` before arriving at `write(s::IO, a::Array{T}) where T` which does `ccall(jl_uv_write())`.

`jl_uv_write()` calls `uv_write()` to write "Hello World!" to `JL_STDOUT`. See Libuv wrappers for stdio.:

```
Hello World!
```

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to `main()`.

jl_atexit_hook()

`main()` calls `jl_atexit_hook()`. This calls `_atexit` for each module, then calls `jl_gc_run_all_finalizers()` and cleans up libuv handles.

julia_save()

Finally, `main()` calls `julia_save()`, which if requested on the command line, saves the runtime state to a new system image. See `jl_compile_all()` and `jl_save_system_image()`.

99.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the `Meta.parse` function), and manipulated by macros. It is a structured representation of code as it is written, constructed by `julia-parser.scm` from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by `julia-syntax.scm`.

First we will focus on the AST, since it is needed to write macros.

Surface syntax AST

Front end ASTs consist almost entirely of `Expr`s and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each

parenthesized list corresponds to an Expr, where the first element is the head. For example `(call f x)` corresponds to `Expr(:call, :f, :x)` in Julia.

Calls

`do` syntax:

```
f(x) do a,b
    body
end
```

parses as `(do (call f x) (-> (tuple a b) (block body)))`.

Operators

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as "syntactic operators". Some operators (+ and *) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Bracketed forms

Macros

Strings

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Imports and such

`using` has the same representation as `import`, but with expression head `:using` instead of `:import`.

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else
    e
end
```

parses as:

```
(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
            (block (line 5 e))))
```

A `while` loop parses as `(while condition body)`.

A `for` loop parses as `(for (= var iter) body)`. If there is more than one iteration specification, they are parsed as a block: `(for (block (= v1 iter1) (= v2 iter2)) body)`.

`break` and `continue` are parsed as 0-argument expressions `(break)` and `(continue)`.

`let` is parsed as `(let (= var val) body)` or `(let (block (= var1 val1) (= var2 val2) ...) body)`, like `for` loops.

A basic function definition is parsed as `(function (call f x) body)`. A more complex example:

```
function f(x::T; k = 1) where T
    return x+1
end
```

parses as:

```
(function (where (call f (parameters (kw k 1))
                         (:: x T))
                 T)
          (block (line 2) (return (call + x 1))))
```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(struct true (curly Foo (<: T S))
        (block (line 2) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as `(try try_block var catch_block finally_block)`. If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

Quote expressions

Julia source syntax forms for code quoting (`quote` and `:( )`) support interpolation with `$`. In Lisp terminology, this means they are actually "backquote" or "quasiquote" forms. Internally, there is also a need for code quoting without interpolation. In Julia's scheme code, non-interpolating quote is represented with the expression head `inert`.

`inert` expressions are converted to Julia `QuoteNode` objects. These objects wrap a single value of any type, and when evaluated simply return that value.

A `quote` expression whose argument is an atom also gets converted to a `QuoteNode`.

Line numbers

Source location information is represented as `(line line_num file_name)` where the third component is optional (and omitted when the current line number, but not file name, changes).

These expressions are represented as `LineNumberNode`s in Julia.

Macros

Macro hygiene is represented through the expression head pair `escape` and `hygienic-scope`. The result of a macro expansion is automatically wrapped in `(hygienic-scope block module)`, to represent the result of the new scope. The user can insert `(escape block)` inside to interpolate code from the caller.

Lowered form

Lowered form (IR) is more important to the compiler, since it is used for type inference, optimizations like inlining, and code generation. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

In addition to `Symbol`s and some number types, the following data types exist in lowered form:

- `Expr`

  Has a node type indicated by the `head` field, and an `args` field which is a `Vector{Any}` of subexpressions. While almost every part of a surface AST is represented by an `Expr`, the IR uses only a limited number of `Expr`s, mostly for calls, conditional branches (`gotoifnot`), and returns.

- `Slot`

  Identifies arguments and local variables by consecutive numbering. `Slot` is an abstract type with subtypes `SlotNumber` and `TypedSlot`. Both types have an integer-valued `id` field giving the slot index. Most slots have the same type at all uses, and so are represented with `SlotNumber`. The types of these slots are found in the `slottypes` field of their `MethodInstance` object. Slots that require per-use type annotations are represented with `TypedSlot`, which has a `typ` field.

- `CodeInfo`

  Wraps the IR of a group of statements. Its `code` field is an array of expressions to execute.

- `GotoNode`

  Unconditional branch. The argument is the branch target, represented as an index in the code array to jump to.

- `QuoteNode`

  Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a `QuoteNode` whose `value` field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

- `GlobalRef`

  Refers to global variable `name` in module `mod`.

- `SSAValue`

  Refers to a consecutively-numbered (starting at 1) static single assignment (SSA) variable inserted by the compiler. The number (`id`) of an `SSAValue` is the code array index of the expression whose value it represents.

- `NewvarNode`

  Marks a point where a variable (slot) is created. This has the effect of resetting a variable to undefined.

`Expr` types

These symbols appear in the `head` field of `Expr`s in lowered form.

- `call`

  Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

- `invoke`

  Function call (static dispatch). `args[1]` is the MethodInstance to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

- `static_parameter`

  Reference a static parameter by index.

- `gotoifnot`

  Conditional branch. If `args[1]` is false, goes to the index identified in `args[2]`.

- `=`

  Assignment. In the IR, the first argument is always a Slot or a GlobalRef.

- `method`

  Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 3-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

  The 3-argument form has the following arguments:

  - `args[1]`

    A function name, or `false` if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is `false`, it means a method is being added strictly by type, `(::T)(x) = x`.

  - `args[2]`

    A `SimpleVector` of argument type data. `args[2][1]` is a `SimpleVector` of the argument types, and `args[2][2]` is a `SimpleVector` of type variables corresponding to the method's static parameters.

  - `args[3]`

    A `CodeInfo` of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a `:lambda` expression.

- `struct_type`

  A 7-argument expression that defines a new `struct`:

  - `args[1]`

    The name of the `struct`

  - `args[2]`

    A `call` expression that creates `SimpleVector` specifying its parameters

  - `args[3]`

    A `call` expression that creates `SimpleVector` specifying its fieldnames

  - `args[4]`

    A `Symbol` or `GlobalRef` specifying the supertype (e.g., `:Integer` or `GlobalRef(Core, :Any)`)

  - `args[5]`

    A `call` expression that creates `SimpleVector` specifying its fieldtypes

  - `args[6]`

    A Bool, true if `mutable`

  - `args[7]`

    The number of arguments to initialize. This will be the number of fields, or the minimum number of fields called by an inner constructor's `new` statement.

- `abstract_type`

  A 3-argument expression that defines a new abstract type. The arguments are the same as the first three arguments of `struct_type` expressions.

- `primitive_type`

  A 4-argument expression that defines a new primitive type. Arguments 1, 2, and 4 are the same as `struct_type`. Argument 3 is the number of bits.

- `global`

  Declares a global binding.

- `const`

  Declares a (global) variable as constant.

- `new`

  Allocates a new struct-like object. First argument is the type. The `new` pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary `new` expressions can easily segfault.

- `splatnew`

  Similar to `new`, except field values are passed as a single tuple. Works similarly to `Base.splat(new)` if `new` were a first-class function, hence the name.

- `return`

  Returns its argument as the value of the enclosing function.

- `isdefined`

  `Expr(:isdefined, :x)` returns a Bool indicating whether `x` has already been defined in the current scope.

- `the_exception`

  Yields the caught exception inside a `catch` block, as returned by `jl_current_exception()`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error. Yields a token which is consumed by `pop_exception`.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `pop_exception`

  Pop the stack of current exceptions back to the state at the associated `enter` when leaving a catch block. `args[1]` contains the token from the associated `enter`.

  > Julia 1.1
  >
  > `pop_exception` is new in Julia 1.1.

- `inbounds`

  Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

- `boundscheck`

  Has the value `false` if inlined into a section of code marked with `@inbounds`, otherwise has the value `true`.

- `loopinfo`

  Marks the end of the a loop. Contains metadata that is passed to `LowerSimdLoop` to either mark the inner loop of `@simd` expression, or to propagate information to LLVM loop passes.

- `copyast`

  Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

- `meta`

  Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

  - `:inline` and `:noinline`: Inlining hints.

Method

A unique'd container describing the shared metadata for a single method.

- `name`, `module`, `file`, `line`, `sig`

  Metadata to uniquely identify the method for the computer and the human.

- `ambig`

  Cache of other methods that may be ambiguous with this one.

- `specializations`

  Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.

- `source`

  The original source code (if available, usually compressed).

- `generator`

  A callable object which can be executed to get specialized source for a specific method signature.

- `roots`

  Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

- `nargs`, `isva`, `called`, `isstaged`, `pure`

  Descriptive bit-fields for the source code of this Method.

- `primary_world`

  The world age that "owns" this Method.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- `specTypes`

  The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

- `def`

  The `Method` that this function describes a specialization of. Or a `Module`, if this is a top-level Lambda expanded in Module, and which is not part of a Method.

- `sparam_vals`

  The values of the static parameters in `specTypes` indexed by `def.sparam_syms`. For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

- `uninferred`

  The uncompressed source code for a toplevel thunk. Additionally, for a generated function, this is one of many places that the source code might be found.

- `backedges`

  We store the reverse-list of cache dependencies for efficient tracking of incremental reanalysis/recompilation work that may be needed after a new method definitions. This works by keeping a list of the other `MethodInstance` that have been inferred or optimized to contain a possible call to this `MethodInstance`. Those optimization results might be stored somewhere in the `cache`, or it might have been the result of something we didn't want to cache, such as constant propagation. Thus we merge all of those backedges to various cache entries here (there's almost always only the one applicable cache entry with a sentinal value for max_world anyways).

- `cache`

  Cache of `CodeInstance` objects that share this template instantiation.

CodeInstance

- `def`

  The `MethodInstance` that this cache entry is derived from.

- `rettype/rettype_const`

  The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

- `inferred`

  May contain a cache of the inferred source for this function, or it could be set to `nothing` to just indicate `rettype` is inferred.

- `ftpr`

  The generic jlcall entry point.

- `jlcall_api`

  The ABI to use when calling `fptr`. Some significant ones include:

  - 0 - Not compiled yet
  - 1 - JLCALLABLE ‘jlvaluet ()(jlfunctiont *f, jlvaluet *args[nargs], uint32t nargs)‘
  - 2 - Constant (value stored in `rettype_const`)
  - 3 - With Static-parameters forwarded `jl_value_t *(*)(jl_svec_t *sparams, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 4 - Run in interpreter `jl_value_t *(*)(jl_method_instance_t *meth, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

- `min_world` / `max_world`

  The range of world ages for which this method instance is valid to be called. If max_world is the special token value `-1`, the value is not yet known. It may continue to be used until we encounter a backedge that requires us to reconsider.

CodeInfo

A (usually temporary) container for holding lowered source code.

- `code`

  An `Any` array of statements

- `slotnames`

  An array of symbols giving names for each slot (argument or local variable).

- `slotflags`

  A `UInt8` array of slot properties, represented as bit flags:

  - 2 - assigned (only false if there are no assignment statements with this var on the left)
  - 8 - const (currently unused for local variables)
  - 16 - statically assigned once
  - 32 - might be used before assigned. This flag is only valid after type inference.

- `ssavaluetypes`

  Either an array or an `Int`.

  If an `Int`, it gives the number of compiler-inserted temporary locations in the function (the length of `code` array). If an array, specifies a type for each location.

- `ssaflags`

  Statement-level flags for each expression in the function. Many of these are reserved, but not yet implemented:

  - 0 = inbounds
  - 1,2 = <reserved> inlinehint,always-inline,noinline
  - 3 = <reserved> strict-ieee (strictfp)
  - 4-6 = <unused>
  - 7 = <reserved> has out-of-band info

- `linetable`

  An array of source location objects

- `codelocs`

  An array of integer indices into the `linetable`, giving the location associated with each statement.

Optional Fields:

- `slottypes`

  An array of types for the slots.

- `rettype`

  The inferred return type of the lowered form (IR). Default value is `Any`.

- `method_for_inference_limit_heuristics`

  The `method_for_inference_heuristics` will expand the given method's generator if necessary during inference.

- `parent`

  The `MethodInstance` that "owns" this object (if applicable).

- `min_world/max_world`

  The range of world ages for which this code was valid at the time when it had been inferred.

Boolean properties:

- `inferred`

  Whether this has been produced by type inference.

- `inlineable`

  Whether this should be eligible for inlining.

- `propagate_inbounds`

  Whether this should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

- `pure`

  Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

99.3 More about types

If you've used Julia for a while, you understand the fundamental role that types play. Here we try to get under the hood, focusing particularly on Parametric Types.

Types and sets (and `Any` and `Union{}`/`Bottom`)

It's perhaps easiest to conceive of Julia's type system in terms of sets. While programs manipulate individual values, a type refers to a set of values. This is not the same thing as a collection; for example a `Set` of values is itself a single `Set` value. Rather, a type describes a set of possible values, expressing uncertainty about which value we have.

A concrete type `T` describes the set of values whose direct tag, as returned by the `typeof` function, is `T`. An abstract type describes some possibly-larger set of values.

`Any` describes the entire universe of possible values. `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other concrete types. Internally, Julia also makes heavy use of another type known as `Bottom`, which can also be written as `Union{}`. This corresponds to the empty set.

Julia's types support the standard operations of set theory: you can ask whether `T1` is a "subset" (subtype) of `T2` with `T1 <: T2`. Likewise, you intersect two types using `typeintersect`, take their union with `Union`, and compute a type that contains their union with `typejoin`:

```
julia> typeintersect(Int, Float64)
Union{}

julia> Union{Int, Float64}
Union{Float64, Int64}

julia> typejoin(Int, Float64)
Real

julia> typeintersect(Signed, Union{UInt8, Int8})
Int8

julia> Union{Signed, Union{UInt8, Int8}}
Union{UInt8, Signed}

julia> typejoin(Signed, Union{UInt8, Int8})
Integer

julia> typeintersect(Tuple{Integer,Float64}, Tuple{Int,Real})
Tuple{Int64,Float64}

julia> Union{Tuple{Integer,Float64}, Tuple{Int,Real}}
Union{Tuple{Int64,Real}, Tuple{Integer,Float64}}

julia> typejoin(Tuple{Integer,Float64}, Tuple{Int,Real})
Tuple{Integer,Real}
```

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that methods be sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to `<:`, but with differences that will be discussed below.

UnionAll types

Julia's type system can also express an iterated union of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, Array has two parameters as in `Array{Int,2}`. If we did not know the element type, we could write `Array{T,2} where T`, which is the union of `Array{T,2}` for all values of `T`: `Union{Array{Int8,2}, Array{Int16,2}, ...}`.

Such a type is represented by a `UnionAll` object, which contains a variable (`T` in this example, of type `TypeVar`), and a wrapped type (`Array{T,2}` in this example).

Consider the following methods:

```
f1(A::Array) = 1
f2(A::Array{Int}) = 2
f3(A::Array{T}) where {T<:Any} = 3
f4(A::Array{Any}) = 4
```

The signature - as decribed in Function calls - of `f3` is a `UnionAll` type wrapping a tuple type: `Tuple{typeof(f3), Array{T}} where T`. All but `f4` can be called with `a = [1,2]`; all but `f2` can be called with `b = Any[1,2]`.

Let's look at these types a little more closely:

```
julia> dump(Array)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Core.TypeofBottom Union{}
    ub: Any
  body: UnionAll
    var: TypeVar
      name: Symbol N
      lb: Core.TypeofBottom Union{}
      ub: Any
    body: Array{T,N} <: DenseArray{T,N}
```

This indicates that `Array` actually names a `UnionAll` type. There is one `UnionAll` type for each parameter, nested. The syntax `Array{Int,2}` is equivalent to `Array{Int}{2}`; internally each `UnionAll` is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters; `Array{Int}` gives a type equivalent to `Array{Int,N} where N`.

A `TypeVar` is not itself a type, but rather should be considered part of the structure of a `UnionAll` type. Type variables have lower and upper bounds on their values (in the fields `lb` and `ub`). The symbol `name` is purely cosmetic. Internally, `TypeVar`s are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct `TypeVar`s manually:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

There are convenience versions that allow you to omit any of these arguments except the `name` symbol.

The syntax `Array{T} where T<:Integer` is lowered to

```
let T = TypeVar(:T,Integer)
    UnionAll(T, Array{T})
end
```

so it is seldom necessary to construct a `TypeVar` manually (indeed, this is to be avoided).

Free variables

The concept of a free type variable is extremely important in the type system. We say that a variable `V` is free in type `T` if `T` does not contain the `UnionAll` that introduces variable `V`. For example, the type `Array{Array{V} where V<:Integer}` has no free variables, but the `Array{V}` part inside of it does have a free variable, `V`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T}} where T`, which refers to all homogeneous arrays of arrays. The inner type `Array{T}`, seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the same array type, so `Array{T}` cannot refer to just any old array. One could say that `Array{T}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `jl_has_free_typevars` in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

TypeNames

The following two `Array` types are functionally equivalent, yet print differently:

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)
```

```
julia> Array
Array

julia> Array{TV,NV}
Array{T,N}
```

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```
julia> dump(Array{Int,1}.name)
TypeName
  name: Symbol Array
  module: Module Core
  names: empty SimpleVector
  wrapper: UnionAll
    var: TypeVar
      name: Symbol T
      lb: Core.TypeofBottom Union{}
      ub: Any
    body: UnionAll
      var: TypeVar
        name: Symbol N
        lb: Core.TypeofBottom Union{}
        ub: Any
      body: Array{T,N} <: DenseArray{T,N}
  cache: SimpleVector
    ...

  linearcache: SimpleVector
    ...

  hash: Int64 -7900426068641098781
  mt: MethodTable
    name: Symbol Array
    defs: Nothing nothing
    cache: Nothing nothing
    max_args: Int64 0
    kwsorter: #undef
    module: Module Core
```

```
    : Int64 0
    : Int64 0
```

In this case, the relevant field is `wrapper`, which holds a reference to the top-level type used to make new `Array` types.

```
julia> pointer_from_objref(Array)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array.body.body.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array{TV,NV})
Ptr{Cvoid} @0x00007fcc80c4d930

julia> pointer_from_objref(Array{TV,NV}.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850
```

The `wrapper` field of `Array` points to itself, but for `Array{TV,NV}` it points back to the original definition of the type.

What about the other fields? `hash` assigns an integer to each type. To examine the `cache` field, it's helpful to pick a type that is less heavily used than Array. Let's first create our own type:

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64,2}

julia> MyType{Float32, 5}
MyType{Float32,5}

julia> MyType.body.body.name.cache
svec(MyType{Int64,2}, MyType{Float32,5}, #undef, #undef, #undef, #undef, #undef, #undef)
```

(The cache is pre-allocated to have length 8, but only the first two entries are populated.) Consequently, when you instantiate a parametric type, each concrete type gets saved in a type cache. However, instances containing free type variables are not cached.

Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like `x::Tuple`, the type has to be able to accommodate any tuple. Let's check the parameters:

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any,N} where N)
```

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64,Float64}
```

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{T}} where T, Tuple{Int,Float64})
Union{}
```

Notice that when `T` is free with respect to the `Tuple` type (i.e. its binding `UnionAll` type is outside the `Tuple` type), only one `T` value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `Tuple{}` is distinct:

```
julia> Tuple{}
Tuple{}

julia> Tuple{}.parameters
svec()

julia> typeintersect(Tuple{}, Tuple{Int})
Union{}
```

What is the "primary" tuple-type?

```
julia> pointer_from_objref(Tuple)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{})
Ptr{Cvoid} @0x00007f5998a570d0

julia> pointer_from_objref(Tuple.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{}.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370
```

so `Tuple == Tuple{Vararg{Any}}` is indeed the primary type.

Diagonal types

Consider the type `Tuple{T,T} where T`. A method with this signature would look like:

```
f(x::T, y::T) where {T} = ...
```

According to the usual interpretation of a `UnionAll` type, this `T` ranges over all types, including `Any`, so this type should be equivalent to `Tuple{Any,Any}`. However, this interpretation causes some practical problems.

First, a value of `T` needs to be available inside the method definition. For a call like `f(1, 1.0)`, it's not clear what `T` should be. It could be `Union{Int,Float64}`, or perhaps `Real`. Intuitively, we expect the declaration `x::T` to mean `T === typeof(x)`. To make sure that invariant holds, we need `typeof(x) === typeof(y) === T` in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of `Tuple{T,T} where T`. To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only `Tuple` and `Union` types occur between an occurrence of a variable and the `UnionAll` type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`, where `T` ranges over all concrete types. This gives rise to some interesting subtyping results. For example `Tuple{Real,Real}` is not a subtype of `Tuple{T,T} where T`, because it includes some types like `Tuple{Int8,Int16}` where the two elements

have different types. `Tuple{Real,Real}` and `Tuple{T,T} where T` have the non-trivial intersection `Tuple{T,T} where T<:Real`. However, `Tuple{Real}` is a subtype of `Tuple{T} where T`, because in that case `T` occurs only once and so is not diagonal.

Next consider a signature like the following:

```
f(a::Array{T}, x::T, y::T) where {T} = ...
```

In this case, `T` occurs in invariant position inside `Array{T}`. That means whatever type of array is passed unambiguously determines the value of `T` – we say `T` has an equality constraint on it. Therefore in this case the diagonal rule is not really necessary, since the array determines `T` and we can then allow `x` and `y` to be of any subtypes of `T`. So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial –- some feel this definition should be written as

```
f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

to clarify whether `x` and `y` need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of `y` if `x` and `y` can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
f(x::Union{Nothing,T}, y::T) where {T} = ...
```

Consider what this declaration means. `y` has type `T`. `x` then can have either the same type `T`, or else be of type `Nothing`. So all of the following calls should match:

```
f(1, 1)
f("", "")
f(2.0, 2.0)
f(nothing, 1)
f(nothing, "")
f(nothing, 2.0)
```

These examples are telling us something: when `x` is `nothing::Nothing`, there are no extra constraints on `y`. It is as if the method signature had `y::Any`. This means that whether a variable is diagonal is not a static property based on where it appears in a type. Rather, it depends on where a variable appears when the subtyping algorithm uses it. When `x` has type `Nothing`, we don't need to use the `T` in `Union{Nothing,T}`, so `T` does not "occur". Indeed, we have the following type equivalence:

```
(Tuple{Union{Nothing,T},T} where T) == Union{Tuple{Nothing,Any}, Tuple{T,T} where T}
```

Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters `occurs_inv` and `occurs_cov` (in `src/subtype.c`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `occurs_inv == 0 && occurs_cov > 1`.

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a `UnionAll` type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like `AbstractArray` cannot be a subtype of a concrete type, but a concrete type like `Int` can be, and the empty type `Bottom` can be as well. If a lower bound fails this test the algorithm stops with the answer `false`.

For example, in the problem `Tuple{Int,String} <: Tuple{T,T} where T`, we derive that this would be true if `T` were a supertype of `Union{Int,String}`. However, `Union{Int,String}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `is_leaf_bound`. Note that this test is slightly different from `jl_is_leaf_type`, since it also returns `true` for `Bottom`. Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to the concrete type `Vector{Int}` only if both the upper and lower bounds of `T` equal `Int`. We have not yet worked out a complete algorithm for this.

Introduction to the internal machinery

Most operations for dealing with types are found in the files `jltypes.c` and `subtype.c`. A good way to start is to watch subtyping in action. Build Julia with `make debug` and fire up Julia within a debugger. gdb debugging tips has some tips which may be useful.

Because the subtyping code is used heavily in the REPL itself—and hence breakpoints in this code get triggered often—it will be easiest if you make the following definition:

```
julia> function mysubtype(a,b)
           ccall(:jl_breakpoint, Cvoid, (Any,), nothing)
           a <: b
       end
```

and then set a breakpoint in `jl_breakpoint`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
mysubtype(Tuple{Int,Float64}, Tuple{Integer,Real})
```

We can make it more interesting by trying a more complex case:

```
mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

Subtyping and method sorting

The `type_morespecific` functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if `a` is more specific than `b`, then `a` does not equal `b` and `b` is not more specific than `a`.

If `a` is a strict subtype of `b`, then it is automatically considered more specific. From there, `type_morespecific` employs some less formal rules. For example, `subtype` is sensitive to the number of arguments, but `type_morespecific` may not be. In particular, `Tuple{Int,AbstractFloat}` is more specific than `Tuple{Integer}`, even though it is not a subtype. (Of `Tuple{Int,AbstractFloat}` and `Tuple{Integer,Float64}`, neither is more specific than the other.) Likewise, `Tuple{Int,Vararg{Int}}` is not a subtype of `Tuple{Integer}`, but it is considered more specific. However, `morespecific` does get a bonus for length: in particular, `Tuple{Int,Int}` is more specific than `Tuple{Int,Vararg{Int}}`.

If you're debugging how methods get sorted, it can be convenient to define the function:

```
type_morespecific(a, b) = ccall(:jl_type_morespecific, Cint, (Any,Any), a, b)
```

which allows you to test whether tuple type `a` is more specific than tuple type `b`.

99.4 Memory layout of Julia Objects

Object layout (`jl_value_t`)

The `jl_value_t` struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

```
typedef struct jl_value_t* jl_pvalue_t;
```

Each `jl_value_t` struct is contained in a `jl_typetag_t` struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

```
typedef struct {
    opaque metadata;
    jl_value_t value;
} jl_typetag_t;
```

The type of any Julia object is an instance of a leaf `jl_datatype_t` object. The `jl_typeof()` function can be used to query for it:

```
jl_value_t *jl_typeof(jl_value_t *v);
```

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

```
jl_value_t *jl_get_nth_field_checked(jl_value_t *v, size_t i);
jl_value_t *jl_get_field(jl_value_t *o, char *fld);
```

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

```
jl_value_t *v = value->fieldptr[n];
```

As an example, a "boxed" `uint16_t` is stored as follows:

```
struct {
    opaque metadata;
    struct {
        uint16_t data;        // -- 2 bytes
    } jl_value_t;
};
```

This object is created by `jl_box_uint16()`. Note that the `jl_value_t` pointer references the data portion, not the metadata at the top of the struct.

A value may be stored "unboxed" in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The "egal" test (corresponding to the `===` function in Julia), should instead be used to compare two unknown objects for equivalence:

```
int jl_egal(jl_value_t *a, jl_value_t *b);
```

This optimization should be relatively transparent to the API, since the object will be "boxed" on-demand, whenever a `jl_value_t` pointer is needed.

Note that modification of a `jl_value_t` pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

```
int jl_is_mutable(jl_value_t *v);
```

If the object being stored is a `jl_value_t`, the Julia garbage collector must be notified also:

```
void jl_gc_wb(jl_value_t *parent, jl_value_t *ptr);
```

However, the Embedding Julia section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are defined in `julia.h`. The corresponding global `jl_datatype_t` objects are created by `jl_init_types` in `jltypes.c`.

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `jl_typetag_t` to track each object in the system. Further details about this algorithm can be found in the comments of the garbage collector implementation in `gc.c`.

Object allocation

Most new objects are allocated by `jl_new_structv()`:

```
jl_value_t *jl_new_struct(jl_datatype_t *type, ...);
jl_value_t *jl_new_structv(jl_datatype_t *type, jl_value_t **args, uint32_t na);
```

Although, `isbits` objects can be also constructed directly from memory:

```
jl_value_t *jl_new_bits(jl_value_t *bt, void *data)
```

And some objects have special constructors that must be used instead of the above functions:

Types:

```
jl_datatype_t *jl_apply_type(jl_datatype_t *tc, jl_tuple_t *params);
jl_datatype_t *jl_apply_array_type(jl_datatype_t *type, size_t dim);
```

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in `julia.h`. These are used in `jl_init_types()` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

```
jl_tuple_t *jl_tuple(size_t n, ...);
jl_tuple_t *jl_tuplev(size_t n, jl_value_t **v);
jl_tuple_t *jl_alloc_tuple(size_t n);
```

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a `Base.tuple()` object may be an array of pointers to the objects contained by the tuple equivalent to:

```
typedef struct {
    size_t length;
    jl_value_t *data[length];
} jl_tuple_t;
```

However, in other cases, the tuple may be converted to an anonymous `isbits` type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a `jl_value_t*`).

Symbols:

```
jl_sym_t *jl_symbol(const char *str);
```

Functions and MethodInstance:

```
jl_function_t *jl_new_generic_function(jl_sym_t *name);
jl_method_instance_t *jl_new_method_instance(jl_value_t *ast, jl_tuple_t *sparams);
```

Arrays:

```
jl_array_t *jl_new_array(jl_value_t *atype, jl_tuple_t *dims);
jl_array_t *jl_new_arrayv(jl_value_t *atype, ...);
jl_array_t *jl_alloc_array_1d(jl_value_t *atype, size_t nr);
jl_array_t *jl_alloc_array_2d(jl_value_t *atype, size_t nr, size_t nc);
jl_array_t *jl_alloc_array_3d(jl_value_t *atype, size_t nr, size_t nc, size_t z);
jl_array_t *jl_alloc_vec_any(size_t n);
```

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the `julia.h` header file.

Internal to Julia, storage is typically allocated by `newstruct()` (or `newobj()` for the special types):

```
jl_value_t *newstruct(jl_value_t *type);
jl_value_t *newobj(jl_value_t *type, size_t nfields);
```

And at the lowest level, memory is getting allocated by a call to the garbage collector (in `gc.c`), then tagged with its type:

```
jl_value_t *jl_gc_allocobj(size_t nbytes);
void jl_set_typeof(jl_value_t *v, jl_datatype_t *type);
```

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with `malloc()` for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. `nothing::Nothing`.

See Singleton Types and Nothingness and missing values

99.5 Eval of Julia code

One of the hardest parts about learning how the Julia Language runs code is learning how all of the pieces work together to execute a block of code.

Each chunk of code typically makes a trip through many steps with potentially unfamiliar names, such as (in no particular order): flisp, AST, C++, LLVM, `eval`, `typeinf`, `macroexpand`, sysimg (or system image), bootstrapping, compile, parse, execute, JIT, interpret, box, unbox, intrinsic function, and primitive function, before turning into the desired result (hopefully).

Definitions

- REPL

  REPL stands for Read-Eval-Print Loop. It's just what we call the command line environment for short.

- AST

  Abstract Syntax Tree The AST is the digital representation of the code structure. In this form the code has been tokenized for meaning so that it is more suitable for manipulation and execution.

Julia Execution

The 10,000 foot view of the whole process is as follows:

1. The user starts `julia`.
2. The C function `main()` from `ui/repl.c` gets called. This function processes the command line arguments, filling in the `jl_options` struct and setting the variable `ARGS`. It then initializes Julia (by calling `julia_init` in `task.c`, which may load a previously compiled sysimg). Finally, it passes off control to Julia by calling `Base._start()`.
3. When `_start()` takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.
4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.

5. If the block of code to run is in a file, `jl_load(char *filename)` gets invoked to load the file and parse it. Each fragment of code is then passed to `eval` to execute.
6. Each fragment of code (or AST), is handed off to `eval()` to turn into results.
7. `eval()` takes each code fragment and tries to run it in `jl_toplevel_eval_flex()`.
8. `jl_toplevel_eval_flex()` decides whether the code is a "toplevel" action (such as `using` or `module`), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.
9. `jl_toplevel_eval_flex()` then expands the code to eliminate any macros and to "lower" the AST to make it simpler to execute.
10. `jl_toplevel_eval_flex()` then uses some simple heuristics to decide whether to JIT compiler the AST or to interpret it directly.
11. The bulk of the work to interpret code is handled by `eval` in `interpreter.c`.
12. If instead, the code is compiled, the bulk of the work is handled by `codegen.cpp`. Whenever a Julia function is called for the first time with a given set of argument types, type inference will be run on that function. This information is used by the codegen step to generate faster code.
13. Eventually, the user quits the REPL, or the end of the program is reached, and the `_start()` method returns.
14. Just before exiting, `main()` calls `jl_atexit_hook(exit_code)`. This calls `Base._atexit()` (which calls any functions registered to `atexit()` inside Julia). Then it calls `jl_gc_run_all_finalizers()`. Finally, it gracefully cleans up all `libuv` handles and waits for them to flush and close.

Parsing

The Julia parser is a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in src/flisp.

The interface functions for this are primarily defined in `jlfrontend.scm`. The code in `ast.c` handles this handoff on the Julia side.

The other relevant files at this stage are `julia-parser.scm`, which handles tokenizing Julia code and turning it into an AST, and `julia-syntax.scm`, which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

Macro Expansion

When `eval()` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `eval()` (in Julia), to the parser function `jl_macroexpand()` (written in `flisp`) to the Julia macro itself (written in - what else - Julia) via `fl_invoke_julia_macro()`, and back.

Typically, macro expansion is invoked as a first step during a call to `Meta.lower()`/`jl_expand()`, although it can also be invoked directly by a call to `macroexpand()`/`jl_macroexpand()`.

Type Inference

Type inference is implemented in Julia by `typeinf()` in `compiler/typeinfer.jl`. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

More Definitions

- JIT

  Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.

- LLVM

  Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.

- C++

  The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.

- box

  This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.

- unbox

  The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).

- generic function

  A Julia function composed of multiple "methods" that are selected for dynamic dispatch based on the argument type-signature

- anonymous function or "method"

  A Julia function without a name and without type-dispatch capabilities

- primitive function

  A function implemented in C but exposed in Julia as a named function "method" (albeit without generic function dispatch capabilities, similar to a anonymous function)

- intrinsic function

  A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to `Core.Intrinsics.box(T, ...)` to reassign type information to the value.

JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to `jl_init_codegen` in `codegen.cpp`.

On demand, a Julia method is converted into a native function by the function `emit_function(jl_method_instance_t*)`. (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `emit_expr()` until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For example, `emit_known_call()` knows how to inline many of the primitive functions (defined in `builtins.c`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- `debuginfo.cpp`

  Handles backtraces for JIT functions

- `ccall.cpp`

  Handles the ccall and llvmcall FFI, along with various `abi_*.cpp` files

- `intrinsics.cpp`

  Handles the emission of various low-level intrinsic functions

  Bootstrapping

  The process of creating a new system image is called "bootstrapping".

  The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

System Image

The system image is a precompiled archive of a set of Julia files. The `sys.ji` file distributed with Julia is one such system image, generated by executing the file `sysimg.jl`, and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the `Main`, `Core`, and `Base` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `jl_save_system_image/jl_restore_system_image` in `staticdata.c`.

If there is no sysimg file (`jl_options.image_file == NULL`), this also implies that `--build` was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal `Core` and `Main` modules are created. Then a file named `boot.jl` is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

99.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

- LLVM ghosts (zero-length types) are omitted.
- LLVM scalars and vectors are passed by value.
- LLVM aggregates (arrays and structs) are passed by reference.

A small return values is returned as LLVM return values. A large return values is returned via the "structure return" (`sret`) convention, where the caller provides a pointer to a return slot.

An argument or return values that is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro `JL_CALLABLE`. The convention uses exactly 3 parameters:

- `F` - Julia representation of function that is being applied

- `args` - pointer to array of pointers to boxes
- `nargs` - length of the array

The return value is a pointer to a box.

C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

99.7 High-level Overview of the Native-Code Generation Process

Representation of Pointers

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call `literal_pointer_val`. It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large `gvals` table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large `fvals` table. Like globals, this allows the deserializer to reference them by index.

Note that `extern` functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that `ccall` functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a `jl_cgval_t` struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: `mark_julia_type` (for immediate values) and `mark_julia_slot` (for pointers to values).

The function `convert_julia_type` can transform between any two types. It returns an R-value with `cgval.typ` set to `typ`. It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

By contrast `update_julia_type` will change `cgval.typ` to `typ`, only if it can be done at zero-cost (i.e. without emitting any code).

Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

- mark-type
- load-local
- store-local
- isa
- is
- emit_typeof
- emit_sizeof
- boxed
- unbox
- specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of `< void* union, byte selector >`. The selector is fixed-size as `byte & 0x7f`, and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the `union*` is actually a tagged heap-allocated `jl_value_t*`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector (`byte & 0x80`) can be tested to determine if the `void*` is actually a heap-allocated (`jl_value_t*`) box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that `byte & 0x7f` is an exact test for the type, if the value can be represented by a tag — it will never be marked `byte = 0x80`. It is not necessary to also test the type-tag when testing `isa`.

The `union*` memory region may be allocated at any size. The only constraint is that it is big enough to contain the data currently specified by `selector`. It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

Specialized Calling Convention Signature Representation

A `jl_returninfo_t` object describes the calling convention details of any callable.

If any of the arguments or return type of a method can be represented unboxed, and the method is not varargs, it'll be given an optimized calling convention signature based on its `specTypes` and `rettype` fields.

The general principles are that:

- Primitive types get passed in int/float registers.
- Tuples of VecElement types get passed in vector registers.
- Structs get passed on the stack.
- Return values are handle similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden sret argument.

The total logic for this is implemented by `get_specsig_function` and `deserves_sret`.

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

99.8 Julia Functions

This document will explain how functions, method definitions, and method tables work.

Method Tables

Every function in Julia is a generic function. A generic function is conceptually a single function, but consists of many definitions, or methods. The methods of a generic function are stored in a method table. Method tables (type `MethodTable`) are associated with `TypeName`s. A `TypeName` describes a family of parameterized types. For example `Complex{Float32}` and `Complex{Float64}` share the same `Complex` type name object.

All objects in Julia are potentially callable, because every object has a type, which in turn has a `TypeName`.

Function calls

Given the call `f(x,y)`, the following steps are performed: first, the method table to use is accessed as `typeof(f).name.mt`. Second, an argument tuple type is formed, `Tuple{typeof(f), typeof(x), typeof(y)}`. Note that the type of the function itself is the first element. This is because the type might have parameters, and so needs to take part in dispatch. This tuple type is looked up in the method table.

This dispatch process is performed by `jl_apply_generic`, which takes two arguments: a pointer to an array of the values f, x, and y, and the number of values (in this case 3).

Throughout the system, there are two kinds of APIs that handle functions and argument lists: those that accept the function and arguments separately, and those that accept a single argument structure. In the first kind of API, the "arguments" part does not contain information about the function, since that is passed separately. In the second kind of API, the function is the first element of the argument structure.

For example, the following function for performing a call accepts just an `args` pointer, so the first element of the args array will be the function to call:

```
jl_value_t *jl_apply(jl_value_t **args, uint32_t nargs)
```

This entry point for the same functionality accepts the function separately, so the `args` array does not contain the function:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs);
```

Adding methods

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. `jl_method_def` implements this operation. `jl_first_argument_datatype` is called to extract the relevant method table from what would be the type of the first argument. This is much more complicated than the corresponding procedure during dispatch, since the argument tuple type might be abstract. For example, we can define:

```
(::Union{Foo{Int},Foo{Int8}})(x) = 0
```

which works since all possible matching methods would belong to the same method table.

Creating generic functions

Since every object is callable, nothing special is needed to create a generic function. Therefore `jl_new_generic_function` simply creates a new singleton (0 size) subtype of `Function` and returns its instance. A function can have a mnemonic

"display name" which is used in debug info and when printing objects. For example the name of `Base.sin` is `sin`. By convention, the name of the created type is the same as the function name, with a # prepended. So `typeof(sin)` is `Base.#sin`.

Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

```
function adder(x)
    return y->x+y
end
```

is lowered to (roughly):

```
struct ##1{T}
    x::T
end

(_::##1)(y) = _.x + y

function adder(x)
    return ##1(x)
end
```

Constructors

A constructor call is just a call to a type. The method table for `Type` contains all constructor definitions. All subtypes of `Type` (`Type`, `UnionAll`, `Union`, and `DataType`) currently share a method table via special arrangement.

Builtins

The "builtin" functions, defined in the `Core` module, are:

```
=== typeof sizeof <: isa typeassert throw tuple getfield setfield! fieldtype
nfields isdefined arrayref arrayset arraysize applicable invoke apply_type _apply
_expr svec
```

These are all singleton objects whose types are subtypes of `Builtin`, which is a subtype of `Function`. Their purpose is to expose entry points in the run time that use the "jlcall" calling convention:

```
jl_value_t *(jl_value_t*, jl_value_t**, uint32_t)
```

The method tables of builtins are empty. Instead, they have a single catch-all method cache entry (`Tuple{Vararg{Any}}`) whose jlcall fptr points to the correct function. This is kind of a hack but works reasonably well.

Keyword arguments

Keyword arguments work by associating a special, hidden function object with each method table that has definitions with keyword arguments. This function is called the "keyword argument sorter" or "keyword sorter", or "kwsorter", and is stored in the `kwsorter` field of `MethodTable` objects. Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single `NamedTuple` argument prepended, which gives the names and values of passed keyword arguments. The kwsorter's job is to move keyword arguments into their canonical positions based on name, plus evaluate and substitute any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another compiler-generated function.

The easiest way to understand the process is to look at how a keyword argument method definition is lowered. The code:

```
function circle(center, radius; color = black, fill::Bool = true, options...)
    # draw
end
```

actually produces three method definitions. The first is a function that accepts all arguments (including keyword arguments) as positional arguments, and includes the code for the method body. It has an auto-generated name:

```
function #circle#1(color, fill::Bool, options, circle, center, radius)
    # draw
end
```

The second method is an ordinary definition for the original `circle` function, which handles the case where no keyword arguments are passed:

```
function circle(center, radius)
    #circle#1(black, true, pairs(NamedTuple()), circle, center, radius)
end
```

This simply dispatches to the first method, passing along default values. `pairs` is applied to the named tuple of rest arguments to provide key-value pair iteration. Note that if the method doesn't accept rest keyword arguments then this argument is absent.

Finally there is the kwsorter definition:

```
function (::Core.kwftype(typeof(circle)))(kws, circle, center, radius)
    if haskey(kws, :color)
        color = kws.color
    else
        color = black
    end
    # etc.

    # put remaining kwargs in `options`
    options = structdiff(kws, NamedTuple{(:color, :fill)})

    # if the method doesn't accept rest keywords, throw an error
    # unless `options` is empty

    #circle#1(color, fill, pairs(options), circle, center, radius)
end
```

The function `Core.kwftype(t)` creates the field `t.name.mt.kwsorter` (if it hasn't been created yet), and returns the type of that function.

This design has the feature that call sites that don't use keyword arguments require no special handling; everything works as if they were not part of the language at all. Call sites that do use keyword arguments are dispatched directly to the called function's kwsorter. For example the call:

```
circle((0,0), 1.0, color = red; other...)
```

is lowered to:

```
kwfunc(circle)(merge((color = red,), other), circle, (0,0), 1.0)
```

`kwfunc` (also in`Core`) fetches the kwsorter for the called function. The keyword splatting operation (written as `other...`) calls the named tuple `merge` function. This function further unpacks each element of `other`, expecting each one to contain two values (a symbol and a value). Naturally, a more efficient implementation is available if all splatted arguments are named tuples. Notice that the original `circle` function is passed through, to handle closures.

Compiler efficiency issues

Generating a new type for every function has potentially serious consequences for compiler resource use when combined with Julia's "specialize on all arguments by default" design. Indeed, the initial implementation of this design

suffered from much longer build and test times, higher memory use, and a system image nearly 2x larger than the baseline. In a naive implementation, the problem is bad enough to make the system nearly unusable. Several significant optimizations were needed to make the design practical.

The first issue is excessive specialization of functions for different values of function-valued arguments. Many functions simply "pass through" an argument to somewhere else, e.g. to another function or to a storage location. Such functions do not need to be specialized for every closure that might be passed in. Fortunately this case is easy to distinguish by simply considering whether a function calls one of its arguments (i.e. the argument appears in "head position" somewhere). Performance-critical higher-order functions like `map` certainly call their argument function and so will still be specialized as expected. This optimization is implemented by recording which arguments are called during the `analyze-variables` pass in the front end. When `cache_method` sees an argument in the `Function` type hierarchy passed to a slot declared as `Any` or `Function`, it behaves as if the `@nospecialize` annotation were applied. This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method cache hash tables. Empirical studies show that the vast majority of dynamically-dispatched calls involve one or two arguments. In turn, many of these cases can be resolved by considering only the first argument. (Aside: proponents of single dispatch would not be surprised by this at all. However, this argument means "multiple dispatch is easy to optimize in practice", and that we should therefore use it, not "we should use single dispatch"!) So the method cache uses the type of the first argument as its primary key. Note, however, that this corresponds to the second element of the tuple type for a function call (the first element being the type of the function itself). Typically, type variation in head position is extremely low — indeed, the majority of functions belong to singleton types with no parameters. However, this is not the case for constructors, where a single method table holds constructors for every type. Therefore the `Type` method table is special-cased to use the first tuple type element instead of the second.

The front end generates type declarations for all closures. Initially, this was implemented by generating normal type declarations. However, this produced an extremely large number of constructors, all of which were trivial (simply passing all arguments through to new). Since methods are partially ordered, inserting all of these methods is O(n^2), plus there are just too many of them to keep around. This was optimized by generating `struct_type` expressions directly (bypassing default constructor generation), and using `new` directly to create closure instances. Not the prettiest thing ever, but you do what you gotta do.

The next problem was the `@test` macro, which generated a 0-argument closure for each test case. This is not really necessary, since each test case is simply run once in place. Therefore, `@test` was modified to expand to a try-catch block that records the test result (true, false, or exception raised) and calls the test suite handler on it.

99.9 Base.Cartesian

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. Most often you can write such algorithms with straightforward techniques; however, there are a few cases where `Base.Cartesian` is still useful or necessary.

Principles of usage

A simple example of usage is:

```
@nloops 3 i A begin
    s += @nref 3 A i
end
```

which generates the following code:

```
for i_3 = axes(A, 3)
    for i_2 = axes(A, 2)
        for i_1 = axes(A, 1)
            s += A[i_1, i_2, i_3]
        end
    end
end
```

In general, Cartesian allows you to write generic code that contains repetitive elements, like the nested loops in this example. Other applications include repeated expressions (e.g., loop unwinding) or creating function calls with variable numbers of arguments without using the "splat" construct (`i...`).

Basic syntax

The (basic) syntax of `@nloops` is as follows:

- The first argument must be an integer (not a variable) specifying the number of loops.
- The second argument is the symbol-prefix used for the iterator variable. Here we used `i`, and variables `i_1, i_2, i_3` were generated.
- The third argument specifies the range for each iterator variable. If you use a variable (symbol) here, it's taken as `axes(A, dim)`. More flexibly, you can use the anonymous-function expression syntax described below.
- The last argument is the body of the loop. Here, that's what appears between the `begin...end`.

There are some additional features of `@nloops` described in the reference section.

`@nref` follows a similar pattern, generating `A[i_1,i_2,i_3]` from `@nref 3 A i`. The general practice is to read from left to right, which is why `@nloops` is `@nloops 3 i A expr` (as in `for i_2 = axes(A, 2)`, where `i_2` is to the left and the range is to the right) whereas `@nref` is `@nref 3 A i` (as in `A[i_1,i_2,i_3]`, where the array comes first).

If you're developing code with Cartesian, you may find that debugging is easier when you examine the generated code, using `@macroexpand`:

```
julia> @macroexpand @nref 2 A i
:(A[i_1, i_2])
```

Supplying the number of expressions

The first argument to both of these macros is the number of expressions, which must be an integer. When you're writing a function that you intend to work in multiple dimensions, this may not be something you want to hard-code. The recommended approach is to use a `@generated function`. Here's an example:

```
@generated function mysum(A::Array{T,N}) where {T,N}
    quote
        s = zero(T)
        @nloops $N i A begin
            s += @nref $N A i
        end
        s
    end
end
```

Naturally, you can also prepare expressions or perform calculations before the `quote` block.

Anonymous-function expressions as macro arguments

Perhaps the single most powerful feature in `Cartesian` is the ability to supply anonymous-function expressions that get evaluated at parsing time. Let's consider a simple example:

```
@nexprs 2 j->(i_j = 1)
```

`@nexprs` generates `n` expressions that follow a pattern. This code would generate the following statements:

```
i_1 = 1
i_2 = 1
```

In each generated statement, an "isolated" `j` (the variable of the anonymous function) gets replaced by values in the range `1:2`. Generally speaking, Cartesian employs a LaTeX-like syntax. This allows you to do math on the index `j`. Here's an example computing the strides of an array:

```
s_1 = 1
@nexprs 3 j->(s_{j+1} = s_j * size(A, j))
```

would generate expressions

```
s_1 = 1
s_2 = s_1 * size(A, 1)
s_3 = s_2 * size(A, 2)
s_4 = s_3 * size(A, 3)
```

Anonymous-function expressions have many uses in practice.

Macro reference `Base.Cartesian.@nloops` – Macro.

```
@nloops N itersym rangeexpr bodyexpr
@nloops N itersym rangeexpr preexpr bodyexpr
@nloops N itersym rangeexpr preexpr postexpr bodyexpr
```

Generate `N` nested loops, using `itersym` as the prefix for the iteration variables. `rangeexpr` may be an anonymous-function expression, or a simple symbol `var` in which case the range is `axes(var, d)` for dimension `d`.

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
@nloops 2 i A d -> j_d = min(i_d, 5) begin
    s += @nref 2 A j
end
```

would generate:

```
for i_2 = axes(A, 2)
    j_2 = min(i_2, 5)
    for i_1 = axes(A, 1)
        j_1 = min(i_1, 5)
        s += A[j_1, j_2]
    end
end
```

If you want just a post-expression, supply `nothing` for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

source

`Base.Cartesian.@nref` – Macro.

```
@nref N A indexexpr
```

Generate expressions like `A[i_1, i_2, ...]`. `indexexpr` can either be an iteration-symbol prefix, or an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nref 3 A i
:(A[i_1, i_2, i_3])
```

source

`Base.Cartesian.@nextract` – Macro.

```
@nextract N esym isym
```

Generate `N` variables `esym_1`, `esym_2`, ..., `esym_N` to extract values from `isym`. `isym` can be either a `Symbol` or anonymous-function expression.

`@nextract 2 x y` would generate

```
x_1 = y[1]
x_2 = y[2]
```

while `@nextract 3 x d->y[2d-1]` yields

```
x_1 = y[1]
x_2 = y[3]
x_3 = y[5]
```

source

`Base.Cartesian.@nexprs` – Macro.

```
@nexprs N expr
```

Generate `N` expressions. `expr` should be an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nexprs 4 i -> y[i] = A[i+j]
quote
    y[1] = A[1 + j]
    y[2] = A[2 + j]
    y[3] = A[3 + j]
    y[4] = A[4 + j]
end
```

source

`Base.Cartesian.@ncall` – Macro.

```
@ncall N f sym...
```

Generate a function call expression. `sym` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into `N` arguments.

For example, `@ncall 3 func a` generates

```
func(a_1, a_2, a_3)
```

while `@ncall 2 func a b i->c[i]` yields

```
func(a, b, c[1], c[2])
```

source

`Base.Cartesian.@ntuple` – Macro.

```
@ntuple N expr
```

Generates an `N`-tuple. `@ntuple 2 i` would generate `(i_1, i_2)`, and `@ntuple 2 k->k+1` would generate `(2,3)`.

source

`Base.Cartesian.@nall` – Macro.

```
@nall N expr
```

Check whether all of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nall 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 && i_2 > 1 && i_3 > 1)`. This can be convenient for bounds-checking.

source

`Base.Cartesian.@nany` – Macro.

```
@nany N expr
```

Check whether any of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nany 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 || i_2 > 1 || i_3 > 1)`.

source

`Base.Cartesian.@nif` – Macro.

```
@nif N conditionexpr expr
@nif N conditionexpr expr elseexpr
```

Generates a sequence of `if ... elseif ... else ... end` statements. For example:

```
@nif 3 d->(i_d >= size(A,d)) d->(error("Dimension ", d, " too big")) d->println("All OK")
```

would generate:

```
if i_1 > size(A, 1)
    error("Dimension ", 1, " too big")
elseif i_2 > size(A, 2)
    error("Dimension ", 2, " too big")
else
    println("All OK")
end
```

source

99.10 Talking to the compiler (the `:meta` mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a `:meta` expression, which is typically (but not necessarily) the first expression in the body of a function.

`:meta` expressions are created with macros. As an example, consider the implementation of the `@inline` macro:

```
macro inline(ex)
    esc(isa(ex, Expr) ? pushmeta!(ex, :inline) : ex)
end
```

Here, `ex` is expected to be an expression defining a function. A statement like this:

```
@inline function myfunction(x)
    x*(x+3)
end
```

gets turned into an expression like this:

```
quote
    function myfunction(x)
        Expr(:meta, :inline)
        x*(x+3)
    end
end
```

`Base.pushmeta!(ex, :symbol, args...)` appends `:symbol` to the end of the `:meta` expression, creating a new `:meta` expression if necessary. If `args` is specified, a nested expression containing `:symbol` and these arguments is appended instead, which can be used to specify additional information.

To use the metadata, you have to parse these `:meta` expressions. If your implementation can be performed within Julia, `Base.popmeta!` is very handy: `Base.popmeta!(body, :symbol)` will scan a function body expression (one without the function signature) for the first `:meta` expression containing `:symbol`, extract any arguments, and return a tuple `(found::Bool, args::Array{Any})`. If the metadata did not have any arguments, or `:symbol` was not found, the `args` array will be empty.

Not yet provided is a convenient infrastructure for parsing `:meta` expressions from C++.

99.11 SubArrays

Julia's `SubArray` type is a container encoding a "view" of a parent `AbstractArray`. This page documents some of the design principles and implementation of `SubArrays`.

One of the major design goals is to ensure high performance for views of both `IndexLinear` and `IndexCartesian` arrays. Furthermore, views of `IndexLinear` arrays should themselves be `IndexLinear` to the extent that it is possible.

Index replacement

Consider making 2d slices of a 3d array:

```
julia> A = rand(2,3,4);

julia> S1 = view(A, :, 1, 2:3)
2×2 view(::Array{Float64,3}, :, 1, 2:3) with eltype Float64:
 0.200586  0.066423
 0.298614  0.956753

julia> S2 = view(A, 1, :, 2:3)
3×2 view(::Array{Float64,3}, 1, :, 2:3) with eltype Float64:
 0.200586  0.066423
 0.246837  0.646691
 0.648882  0.276021
```

`view` drops "singleton" dimensions (ones that are specified by an `Int`), so both `S1` and `S2` are two-dimensional `SubArrays`. Consequently, the natural way to index these is with `S1[i,j]`. To extract the value from the parent array `A`, the natural approach is to replace `S1[i,j]` with `A[i,1,(2:3)[j]]` and `S2[i,j]` with `A[1,i,(2:3)[j]]`.

The key feature of the design of SubArrays is that this index replacement can be performed without any runtime overhead.

SubArray design

Type parameters and fields

The strategy adopted is first and foremost expressed in the definition of the type:

```
struct SubArray{T,N,P,I,L} <: AbstractArray{T,N}
    parent::P
    indices::I
    offset1::Int       # for linear indexing and pointer, only valid when L==true
    stride1::Int       # used only for linear indexing
    ...
end
```

`SubArray` has 5 type parameters. The first two are the standard element type and dimensionality. The next is the type of the parent `AbstractArray`. The most heavily-used is the fourth parameter, a `Tuple` of the types of the indices

for each dimension. The final one, `L`, is only provided as a convenience for dispatch; it's a boolean that represents whether the index types support fast linear indexing. More on that later.

If in our example above `A` is a `Array{Float64, 3}`, our `S1` case above would be a `SubArray{Float64,2,Array{Float64,3},Tuple{Base.Slice{B`
Note in particular the tuple parameter, which stores the types of the indices used to create `S1`. Likewise,

```
julia> S1.indices
(Base.Slice(Base.OneTo(2)), 1, 2:3)
```

Storing these values allows index replacement, and having the types encoded as parameters allows one to dispatch to efficient algorithms.

Index translation

Performing index translation requires that you do different things for different concrete `SubArray` types. For example, for `S1`, one needs to apply the `i,j` indices to the first and third dimensions of the parent array, whereas for `S2` one needs to apply them to the second and third. The simplest approach to indexing would be to do the type-analysis at runtime:

```
parentindices = Vector{Any}()
for thisindex in S.indices
    ...
    if isa(thisindex, Int)
        # Don't consume one of the input indices
        push!(parentindices, thisindex)
    elseif isa(thisindex, AbstractVector)
        # Consume an input index
        push!(parentindices, thisindex[inputindex[j]])
        j += 1
    elseif isa(thisindex, AbstractMatrix)
        # Consume two input indices
        push!(parentindices, thisindex[inputindex[j], inputindex[j+1]])
        j += 2
    elseif ...
end
S.parent[parentindices...]
```

Unfortunately, this would be disastrous in terms of performance: each element access would allocate memory, and involves the running of a lot of poorly-typed code.

The better approach is to dispatch to specific methods to handle each type of stored index. That's what `reindex` does: it dispatches on the type of the first stored index and consumes the appropriate number of input indices, and then it recurses on the remaining indices. In the case of `S1`, this expands to

```
Base.reindex(S1, S1.indices, (i, j)) == (i, S1.indices[2], S1.indices[3][j])
```

for any pair of indices `(i,j)` (except CartesianIndexs and arrays thereof, see below).

This is the core of a `SubArray`; indexing methods depend upon `reindex` to do this index translation. Sometimes, though, we can avoid the indirection and make it even faster.

Linear indexing

Linear indexing can be implemented efficiently when the entire array has a single stride that separates successive elements, starting from some offset. This means that we can pre-compute these values and represent linear indexing simply as an addition and multiplication, avoiding the indirection of `reindex` and (more importantly) the slow computation of the cartesian coordinates entirely.

For `SubArray` types, the availability of efficient linear indexing is based purely on the types of the indices, and does not depend on values like the size of the parent array. You can ask whether a given set of indices supports fast linear indexing with the internal `Base.viewindexing` function:

```
julia> Base.viewindexing(S1.indices)
IndexCartesian()

julia> Base.viewindexing(S2.indices)
IndexLinear()
```

This is computed during construction of the `SubArray` and stored in the `L` type parameter as a boolean that encodes fast linear indexing support. While not strictly necessary, it means that we can define dispatch directly on `SubArray{T,N,A,I,true}` without any intermediaries.

Since this computation doesn't depend on runtime values, it can miss some cases in which the stride happens to be uniform:

```
julia> A = reshape(1:4*2, 4, 2)
4×2 reshape(::UnitRange{Int64}, 4, 2) with eltype Int64:
 1  5
 2  6
 3  7
```

```
 4  8

julia> diff(A[2:2:4,:][:])
3-element Array{Int64,1}:
 2
 2
 2
```

A view constructed as `view(A, 2:2:4, :)` happens to have uniform stride, and therefore linear indexing indeed could be performed efficiently. However, success in this case depends on the size of the array: if the first dimension instead were odd,

```
julia> A = reshape(1:5*2, 5, 2)
5×2 reshape(::UnitRange{Int64}, 5, 2) with eltype Int64:
 1   6
 2   7
 3   8
 4   9
 5  10

julia> diff(A[2:2:4,:][:])
3-element Array{Int64,1}:
 2
 3
 2
```

then `A[2:2:4,:]` does not have uniform stride, so we cannot guarantee efficient linear indexing. Since we have to base this decision based purely on types encoded in the parameters of the `SubArray`, `S = view(A, 2:2:4, :)` cannot implement efficient linear indexing.

A few details

- Note that the `Base.reindex` function is agnostic to the types of the input indices; it simply determines how and where the stored indices should be reindexed. It not only supports integer indices, but it supports non-scalar indexing, too. This means that views of views don't need two levels of indirection; they can simply re-compute the indices into the original parent array!
- Hopefully by now it's fairly clear that supporting slices means that the dimensionality, given by the parameter `N`, is not necessarily equal to the dimensionality of the parent array or the length of the `indices` tuple. Neither

do user-supplied indices necessarily line up with entries in the `indices` tuple (e.g., the second user-supplied index might correspond to the third dimension of the parent array, and the third element in the `indices` tuple).

What might be less obvious is that the dimensionality of the stored parent array must be equal to the number of effective indices in the `indices` tuple. Some examples:

```
A = reshape(1:35, 5, 7) # A 2d parent Array
S = view(A, 2:7)         # A 1d view created by linear indexing
S = view(A, :, :, 1:1)   # Appending extra indices is supported
```

Naively, you'd think you could just set `S.parent = A` and `S.indices = (:,:,1:1)`, but supporting this dramatically complicates the reindexing process, especially for views of views. Not only do you need to dispatch on the types of the stored indices, but you need to examine whether a given index is the final one and "merge" any remaining stored indices together. This is not an easy task, and even worse: it's slow since it implicitly depends upon linear indexing.

Fortunately, this is precisely the computation that `ReshapedArray` performs, and it does so linearly if possible. Consequently, `view` ensures that the parent array is the appropriate dimensionality for the given indices by reshaping it if needed. The inner `SubArray` constructor ensures that this invariant is satisfied.

- `CartesianIndex` and arrays thereof throw a nasty wrench into the `reindex` scheme. Recall that `reindex` simply dispatches on the type of the stored indices in order to determine how many passed indices should be used and where they should go. But with `CartesianIndex`, there's no longer a one-to-one correspondence between the number of passed arguments and the number of dimensions that they index into. If we return to the above example of `Base.reindex(S1, S1.indices, (i, j))`, you can see that the expansion is incorrect for `i, j = CartesianIndex(), CartesianIndex(2,1)`. It should skip the `CartesianIndex()` entirely and return:

```
(CartesianIndex(2,1)[1], S1.indices[2], S1.indices[3][CartesianIndex(2,1)[2]])
```

Instead, though, we get:

```
(CartesianIndex(), S1.indices[2], S1.indices[3][CartesianIndex(2,1)])
```

Doing this correctly would require combined dispatch on both the stored and passed indices across all combinations of dimensionalities in an intractable manner. As such, `reindex` must never be called with `CartesianIndex` indices. Fortunately, the scalar case is easily handled by first flattening the `CartesianIndex` arguments to plain integers. Arrays of `CartesianIndex`, however, cannot be split apart into orthogonal pieces so easily. Before attempting to use `reindex`, `view` must ensure that there are no arrays of `CartesianIndex` in the argument list. If there are, it can simply "punt" by avoiding the `reindex` calculation entirely, constructing a nested `SubArray` with two levels of indirection instead.

99.12 isbits Union Optimizations

In Julia, the `Array` type holds both "bits" values as well as heap-allocated "boxed" values. The distinction is whether the value itself is stored inline (in the direct allocated memory of the array), or if the memory of the array is simply a collection of pointers to objects allocated elsewhere. In terms of performance, accessing values inline is clearly an advantage over having to follow a pointer to the actual value. The definition of "isbits" generally means any Julia type with a fixed, determinate size, meaning no "pointer" fields, see `?isbitstype`.

Julia also supports Union types, quite literally the union of a set of types. Custom Union type definitions can be extremely handy for applications wishing to "cut across" the nominal type system (i.e. explicit subtype relationships) and define methods or functionality on these, otherwise unrelated, set of types. A compiler challenge, however, is in determining how to treat these Union types. The naive approach (and indeed, what Julia itself did pre-0.7), is to simply make a "box" and then a pointer in the box to the actual value, similar to the previously mentioned "boxed" values. This is unfortunate, however, because of the number of small, primitive "bits" types (think `UInt8`, `Int32`, `Float64`, etc.) that would easily fit themselves inline in this "box" without needing any indirection for value access. There are two main ways Julia can take advantage of this optimization as of 0.7: isbits Union fields in types, and isbits Union Arrays.

isbits Union Structs

Julia now includes an optimization wherein "isbits Union" fields in types (`mutable struct`, `struct`, etc.) will be stored inline. This is accomplished by determining the "inline size" of the Union type (e.g. `Union{UInt8, Int16}` will have a size of two bytes, which represents the size needed of the largest Union type `Int16`), and in addition, allocating an extra "type tag byte" (`UInt8`), whose value signals the type of the actual value stored inline of the "Union bytes". The type tag byte value is the index of the actual value's type in the Union type's order of types. For example, a type tag value of `0x02` for a field with type `Union{Nothing, UInt8, Int16}` would indicate that an `Int16` value is stored in the 2 bytes of the field in the structure's memory; a `0x01` value would indicate that a `UInt8` value was stored in the first 1 byte of the 2 bytes of the field's memory. Lastly, a value of `0x00` signals that the `nothing` value will be returned for this field, even though, as a singleton type with a single type instance, it technically has a size of 0. The type tag byte for a type's Union field is stored directly after the field's computed Union memory.

isbits Union Arrays

Julia can now also store "isbits Union" values inline in an Array, as opposed to requiring an indirection box. The optimization is accomplished by storing an extra "type tag array" of bytes, one byte per array element, alongside the bytes of the actual array data. This type tag array serves the same function as the type field case: its value signals the type of the actual stored Union value in the array. In terms of layout, a Julia Array can include extra "buffer" space before and after its actual data values, which are tracked in the `a->offset` and `a->maxsize` fields of the `jl_array_t*` type. The "type tag array" is treated exactly as another `jl_array_t*`, but which shares the same

`a->offset`, `a->maxsize`, and `a->len` fields. So the formula to access an isbits Union Array's type tag bytes is `a->data + (a->maxsize - a->offset) * a->elsize + a->offset`; i.e. the Array's `a->data` pointer is already shifted by `a->offset`, so correcting for that, we follow the data all the way to the max of what it can hold `a->maxsize`, then adjust by `a->ofset` more bytes to account for any present "front buffering" the array might be doing. This layout in particular allows for very efficient resizing operations as the type tag data only ever has to move when the actual array's data has to move.

99.13 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.
- Include a `userimg.jl` file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

The `PackageCompiler.jl` package contains convenient wrapper functions to automate this process.

System image optimized for multiple microarchitectures

The system image can be compiled simultaneously for multiple CPU microarchitectures under the same instruction set architecture (ISA). Multiple versions of the same function may be created with minimum dispatch point inserted into shared functions in order to take advantage of different ISA extensions or other microarchitecture features. The version that offers the best performance will be selected automatically at runtime based on available CPU features.

Specifying multiple system image targets

A multi-microarchitecture system image can be enabled by passing multiple targets during system image compilation. This can be done either with the `JULIA_CPU_TARGET` make option or with the `-C` command line option when running the compilation command manually. Multiple targets are separated by `;` in the option string. The syntax for each target

is a CPU name followed by multiple features separated by `,`. All features supported by LLVM are supported and a feature can be disabled with a `-` prefix. (`+` prefix is also allowed and ignored to be consistent with LLVM syntax). Additionally, a few special features are supported to control the function cloning behavior.

1. `clone_all`

   By default, only functions that are the most likely to benefit from the microarchitecture features will be cloned. When `clone_all` is specified for a target, however, all functions in the system image will be cloned for the target. The negative form `-clone_all` can be used to prevent the built-in heuristic from cloning all functions.

2. `base(<n>)`

   Where `<n>` is a placeholder for a non-negative number (e.g. `base(0)`, `base(1)`). By default, a partially cloned (i.e. not `clone_all`) target will use functions from the default target (first one specified) if a function is not cloned. This behavior can be changed by specifying a different base with the `base(<n>)` option. The `n`th target (0-based) will be used as the base target instead of the default (`0`th) one. The base target has to be either `0` or another `clone_all` target. Specifying a non-`clone_all` target as the base target will cause an error.

3. `opt_size`

   This causes the function for the target to be optimized for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

4. `min_size`

   This causes the function for the target to be optimized for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

As an example, at the time of this writing, the following string is used in the creation of the official `x86_64` Julia binaries downloadable from julialang.org:

```
generic;sandybridge,-xsaveopt,clone_all;haswell,-rdrnd,base(1)
```

This creates a system image with three separate targets; one for a generic `x86_64` processor, one with a `sandybridge` ISA (explicitly excluding `xsaveopt`) that explicitly clones all functions, and one targeting the `haswell` ISA, based off of the `sandybridge` sysimg version, and also excluding `rdrnd`. When a Julia implementation loads the generated sysimg, it will check the host processor for matching CPU capability flags, enabling the highest ISA level possible. Note that the base level (`generic`) requires the `cx16` instruction, which is disabled in some virtualization software and must be enabled for the `generic` target to be loaded. Alternatively, a sysimg could be generated with the target `generic,-cx16` for greater compatibility, however note that this may cause performance and stability problems in some code.

Implementation overview

This is a brief overview of different part involved in the implementation. See code comments for each components for more implementation details.

1. System image compilation

   The parsing and cloning decision are done in `src/processor*`. We currently support cloning of function based on the present of loops, simd instructions, or other math operations (e.g. fastmath, fma, muladd). This information is passed on to `src/llvm-multiversioning.cpp` which does the actual cloning. In addition to doing the cloning and insert dispatch slots (see comments in `MultiVersioning::runOnModule` for how this is done), the pass also generates metadata so that the runtime can load and initialize the system image correctly. A detail description of the metadata is available in `src/processor.h`.

2. System image loading

   The loading and initialization of the system image is done in `src/processor*` by parsing the metadata saved during system image generation. Host feature detection and selection decision are done in `src/processor_*.cpp` depending on the ISA. The target selection will prefer exact CPU name match, larger vector register size, and larget number of features. An overview of this process is in `src/processor.cpp`.

99.14 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia dynamically links against LLVM by default. Build with `USE_LLVM_SHLIB=0` to link statically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/codegen.cpp`.

The `-O` option enables LLVM's Basic Alias Analysis.

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/Versions.make`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
LLVM_VER = 6.0.1
```

Besides the LLVM release numerals, you can also use `LLVM_VER = svn` to build against the latest development version of LLVM.

You can also specify to build a debug version of LLVM, by setting either `LLVM_DEBUG = 1` or `LLVM_DEBUG = Release` in your `Make.user` file. The former will be a fully unoptimized build of LLVM and the latter will produce an optimized build of LLVM. Depending on your needs the latter will suffice and it quite a bit faster. If you use `LLVM_DEBUG = Release` you will also want to set `LLVM_ASSERTIONS = 1` to enable diagonstics for different passes. Only `LLVM_DEBUG = 1` implies that option by default.

Passing options to LLVM

You can pass options to LLVM via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using `bash` syntax:

- `export JULIA_LLVM_ARGS = -print-after-all` dumps IR after each pass.
- `export JULIA_LLVM_ARGS = -debug-only=loop-vectorize` dumps LLVM `DEBUG(...)` diagnostics for loop vectorizer. If you get warnings about "Unknown command line argument", rebuild LLVM with `LLVM_ASSERTIONS = 1`.

Debugging LLVM transformations in isolation

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside `julia` would take too long, or because one wants to take advantage of LLVM's tooling (e.g. bugpoint). To get unoptimized IR for the entire system image, pass the `--output-unopt-bc unopt.bc` option to the system image build process, which will output the unoptimized IR to an `unopt.bc` file. This file can then be passed to LLVM tools as usual. `libjulia` can function as an LLVM pass plugin and can be loaded into LLVM tools, to make julia-specific passes available in this environment. In addition, it exposes the `-julia` meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image, one could do:

```
opt -load libjulia.so -julia -o opt.bc unopt.bc
llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

This system image can then be loaded by `julia` as usual.

Alternatively, you can use `--output-jit-bc jit.bc` to obtain a trace of all IR passed to the JIT. This is useful for code that cannot be run as part of the sysimg generation process (e.g. because it creates unserializable state). However, the resulting `jit.bc` does not include sysimage data, and can thus not be used as such.

It is also possible to dump an LLVM IR module for just one Julia function, using:

```
f, T = +, Tuple{Int,Int} # Substitute your function of interest here
optimize = false
open("plus.ll", "w") do f
    println(f, Base._dump_function(f, T, false, false, false, true, :att, optimize))
end
```

These files can be processed the same way as the unoptimized sysimg IR shown above.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS = -print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

```
jl_value_t *any_unoptimized_call(jl_value_t *, jl_value_t **, int);
```

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the argument array. However, this would betray the SSA nature of the uses at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

```
%bitcast = bitcast @any_unoptimized_call to %jl_value_t *(*)(%jl_value_t *, %jl_value_t *)
call cc 37 %jl_value_t *%bitcast(%jl_value_t *%arg1, %jl_value_t *%arg2)
```

The special `cc 37` annotation marks the fact that this call site is really using the jlcall calling convention. This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI. In the code the calling convention number is represented by the `JLCALL_F_CC` constant. In addition, there is the `JLCALL_CC` calling convention which functions similarly, but omits the first argument.

GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

```
if some_condition()
    #= Use some variables maybe =#
    error("An error occurred")
end
```

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early, the GC root slots used in the deleted block, as well as any values kept alive in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are GC tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces
- Operand Bundles
- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation. This allows us to annotate which pointers are GC-tracked in an optimizer-resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically - it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in `src/codegen_shared.cpp`):

- GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a `jl_value_t*` pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.
- Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.
- Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).
- Pointers loaded from tracked object (currently 13): This is used by arrays, which themselves contain a pointer to the managed data. This data area is owned by the array, but is not a GC-tracked object by itself. The compiler guarantees that as long as this pointer is live, the object that this pointer was loaded from will keep being live.

Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

- 0->{Tracked,Derived,CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.
- Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.
- Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

- Loads whose loaded values is in one of the address spaces
- Stores of a value in one of the address spaces to a location
- Stores to a pointer in one of the address spaces
- Calls for which a value in one of the address spaces is an operand
- Calls in jlcall ABI, for which the argument array contains a value
- Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid `jl_value_t*` pointers). The same is true for return instructions (though note that struct return arguments are allowed to have any of the address spaces). The only allowable use of an address space CalleeRooted pointer is to pass it to a call (which must have an appropriately typed operand).

Further, we disallow `getelementptr` in addrspace Tracked. This is because unless the operation is a noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to addrspace Derived first.

Lastly, we disallow `inttoptr/ptrtoint` instructions in these address spaces. Having these instructions would mean that some `i64` values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using `inttoptr` in address space 0 and then decaying to the appropriate address space afterwards.

Supporting `ccall`

One important aspect missing from the discussion so far is the handling of `ccall`. `ccall` has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

```
A = randn(1024)
ccall(:foo, Cvoid, (Ptr{Float64},), A)
```

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the `ccall`. To understand how this is done, first recall the lowering of the above code:

```
return $(Expr(:foreigncall, :(:foo), Cvoid, svec(Ptr{Float64}), :(:ccall), 1, :($(Expr(:foreigncall,
↪  :(:jl_array_ptr), Ptr{Float64}, svec(Any), :(:ccall), 1, :(A)))), :(A)))
```

The last `:(A)`, is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this `ccall`. We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

```
call void inttoptr (i64 ... to void (double*)*)(double* %5) [ "jl_roots"(%jl_value_t addrspace(10)* %A) ]
```

The GC root placement pass will treat the `jl_roots` operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

Supporting `pointer_from_objref`

`pointer_from_objref` is special because it requires the user to take explicit control of GC rooting. By our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

```
declared %jl_value_t *julia.pointer_from_objref(%jl_value_t addrspace(10)*)
```

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to use the result of this operation to establish a global root - the optimizer may have already dropped the value.

Keeping values alive in the absence of uses

In certain cases it is necessary to keep an object alive, even though there is no compiler-visible use of said object. This may be case for low level code that operates on the memory-representation of an object directly or code that needs to interface with C code. In order to allow this, we provide the following intrinsics at the LLVM level:

```
token @llvm.julia.gc_preserve_begin(...)
void @llvm.julia.gc_preserve_end(token)
```

(The `llvm.` in the name is required in order to be able to use the `token` type). The semantics of these intrinsics are as follows: At any safepoint that is dominated by a `gc_preserve_begin` call, but that is not not dominated by a corresponding `gc_preserve_end` call (i.e. a call whose argument is the token returned by a `gc_preserve_begin` call), the values passed as arguments to that `gc_preserve_begin` will be kept live. Note that the `gc_preserve_begin` still counts as a regular use of those values, so the standard lifetime semantics will ensure that the values will be kept alive before entering the preserve region.

99.15 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines libuv wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;
uv_stream_t *JL_STDOUT;
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
int jl_printf(uv_stream_t *s, const char *format, ...);
int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `ui/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to `write(2)` directly to `STDERR_FILENO`:

```
void jl_safe_printf(const char *str, ...);
```

Interface between JL_STD* and Julia code

`Base.stdin`, `Base.stdout` and `Base.stderr` are bound to the `JL_STD*` libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.stdin`, `Base.stdout` and `Base.stderr`.

`reinit_stdio()` uses `ccall` to retrieve pointers to `JL_STD*` and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((stdin, stdout, stderr)))'
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((stdin, stdout, stderr)))' < /dev/null 2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((stdin, stdout, stderr)))' | cat
Tuple{Base.PipeEndpoint,Base.PipeEndpoint,Base.TTY}
```

The `Base.read` and `Base.write` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
               -> ccall(:jl_uv_write, ...)
  jl_uv.c:          -> int jl_uv_write(uv_stream_t *stream, ...)
                        -> uv_write(uvw, stream, buf, ...)
```

printf() during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In `sys.c`, the `JL_STD*` stream pointers are statically initialized to integer constants: `STD*_FILENO (0, 1 and 2)`. In `jl_uv.c` the `jl_uv_puts()` function checks its `uv_stream_t* stream` argument and calls `fwrite()` if stream is set to `STDOUT_FILENO` or `STDERR_FILENO`.

This allows for uniform use of `jl_printf()` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy `ios.c` library

The `src/support/ios.c` library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

`ios.c` is still used by:

- `src/flisp/*.c`

- `src/dump.c` – for serialization file IO and for memory buffers.
- `src/staticdata.c` – for serialization file IO and for memory buffers.
- `base/iostream.jl` – for file IO (see `base/fs.jl` for libuv equivalent).

Use of `ios.c` in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to `jl_printf()` with a legacy `ios_t` stream.

There is a hack in `ios.h` that makes the `ios_t.bm` field line up with the `uv_stream_t.type` and ensures that the values used for `ios_t.bm` to not overlap with valid `UV_HANDLE_TYPE` values. This allows `uv_stream_t` pointers to point to `ios_t` streams.

This is needed because `jl_printf()` caller `jl_static_show()` is passed an `ios_t` stream by femtolisp's `fl_print()` function. Julia's `jl_uv_puts()` function has special handling for this:

```
if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

99.16 Bounds checking

Like many modern programming languages, Julia uses bounds checking to ensure program safety when accessing arrays. In tight inner loops or other performance critical situations, you may wish to skip these bounds checks to improve runtime performance. For instance, in order to emit vectorized (SIMD) instructions, your loop body cannot contain branches, and thus cannot contain bounds checks. Consequently, Julia includes an `@inbounds(...)` macro to tell the compiler to skip such bounds checks within the given block. User-defined array types can use the `@boundscheck(...)` macro to achieve context-sensitive code selection.

Eliding bounds checks

The `@boundscheck(...)` macro marks blocks of code that perform bounds checking. When such blocks are inlined into an `@inbounds(...)` block, the compiler may remove these blocks. The compiler removes the `@boundscheck` block only if it is inlined into the calling function. For example, you might write the method `sum` as:

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

With a custom array-like type `MyArray` having:

```
@inline getindex(A::MyArray, i::Real) = (@boundscheck checkbounds(A,i); A.data[to_index(i)])
```

Then when `getindex` is inlined into `sum`, the call to `checkbounds(A,i)` will be elided. If your function contains multiple layers of inlining, only `@boundscheck` blocks at most one level of inlining deeper are eliminated. The rule prevents unintended changes in program behavior from code further up the stack.

Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the `@inbounds` and `@boundscheck` declarations. For instance, the default `getindex` methods have the chain `getindex(A::AbstractArray, i::Real)` calls `getindex(IndexStyle(A), A, i)` calls `_getindex(::IndexLinear, A, i)`.

To override the "one layer of inlining" rule, a function may be marked with `Base.@propagate_inbounds` to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

- `checkbounds(A, I...)` which calls
  - `checkbounds(Bool, A, I...)` which calls
    - `checkbounds_indices(Bool, axes(A), I)` which recursively calls
      - `checkindex` for each dimension

Here `A` is the array, and `I` contains the "requested" indices. `axes(A)` returns a tuple of "permitted" indices of `A`.

`checkbounds(A, I...)` throws an error if the indices are invalid, whereas `checkbounds(Bool, A, I...)` returns `false` in that circumstance. `checkbounds_indices` discards any information about the array other than its `axes` tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, `checkindex`: typically,

```
checkbounds_indices(Bool, (IA1, IA...), (I1, I...)) = checkindex(Bool, IA1, I1) &
                                                      checkbounds_indices(Bool, IA, I)
```

so `checkindex` checks a single dimension. All of these functions, including the unexported `checkbounds_indices` have docstrings accessible with `?` .

If you have to customize bounds checking for a specific array type, you should specialize `checkbounds(Bool, A, I...)`. However, in most cases you should be able to rely on `checkbounds_indices` as long as you supply useful `axes` for your array type.

If you have novel index types, first consider specializing `checkindex`, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to `CartesianIndex`), then you may have to consider specializing `checkbounds_indices`.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make `checkbounds` the place to specialize on array type, and try to avoid specializations on index types; conversely, `checkindex` is intended to be specialized only on index type (especially, the last argument).

99.17 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed here):

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

- safepoint

  Note that this lock is acquired implicitly by `JL_LOCK` and `JL_UNLOCK`. use the `_NOGC` variants to avoid that for level 1 locks.
  While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.

- shared_map

- finalizers
- pagealloc
- gcpermlock
- flisp

  flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

- typecache

The following is a level 2 lock:

- Module->lock

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

- Method->writelock

The following is a level 4 lock, which can only recurse to acquire level 1, 2, or 3 locks:

- MethodTable->writelock

No Julia code may be called while holding a lock above this point.

The following is a level 6 lock, which can only recurse to acquire locks at lower levels:

- codegen

The following is an almost root lock (level end-1), meaning only the root look may be held when trying to acquire it:

- typeinf

  this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points

  currently the lock is merged with the codegen lock, since they call each other recursively

The following lock synchronizes IO operation. Be aware that doing any I/O (for example, printing warning messages or debug information) while holding any other lock listed above may result in pernicious and hard-to-find deadlocks. BE VERY CAREFUL!

- iolock
- Individual ThreadSynchronizers locks

  this may continue to be held after releasing the iolock, or acquired without it, but be very careful to never attempt to acquire the iolock while holding it

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

- toplevel

  this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!
  additionally, it's unclear if any code can safely run in parallel with an arbitrary toplevel expression, so it may require all threads to get to a safepoint first

Broken Locks

The following locks are broken:

- toplevel

  doesn't exist right now

  fix: create it

Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache, kwsorter type) : MethodTable->writelock

Type declarations : toplevel lock

Type application : typecache lock

Global variable tables : Module->lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance/CodeInstance updates : Method->writelock, codegen lock

- These are set at construction and immutable:
  - specTypes
  - sparam_vals
  - def
- These are set by `jl_type_infer` (while holding codegen lock):
  - cache
  - rettype
  - inferred

```
* valid ages
```

- `inInference` flag:
  - optimization to quickly avoid recurring into `jl_type_infer` while it is already running
  - actual state (of setting `inferred`, then `fptr`) is protected by codegen lock
- Function pointers:
  - these transition once, from `NULL` to a value, while the codegen lock is held
- Code-generator cache (the contents of `functionObjectsDecls`):
  - these can transition multiple times, but only while the codegen lock is held
  - it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as `rettype`) and assume it is coordinated, unless also holding the codegen lock

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

99.18 Arrays with custom indices

Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A,d)` (and not just `0:size(A,d)-1`, either). To facilitate such computations, Julia supports arrays with arbitrary indices.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia. If you find it more convenient to just force your users to supply traditional arrays where indexing starts at one, you can add

```
Base.require_one_based_indexing(arrays...)
```

where `arrays...` is a list of the array objects that you wish to check for anything that violates 1-based indexing.

Generalizing existing code

As an overview, the steps are:

- replace many uses of `size` with `axes`
- replace `1:length(A)` with `eachindex(A)`, or in some cases `LinearIndices(A)`
- replace explicit allocations like `Array{Int}(undef, size(B))` with `similar(Array{Int}, axes(B))`

These are described in more detail below.

Things to watch out for

Because unconventional indexing breaks many people's assumptions that all arrays start indexing with 1, there is always the chance that using such arrays will trigger errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) || throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
```

```
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1; if `dest` starts at a different index than `src`, there is a chance that this code would trigger a segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

Using `axes` for bounds checks and loop iteration

`axes(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange` objects, specifying the range of valid indices along each dimension of `A`. When `A` has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension `d`, there is `axes(A, d)`.

Base implements a custom range type, `OneTo`, where `OneTo(n)` means the same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new `AbstractArray` type, this is the default returned by `axes`, and it indicates that this array type uses "conventional" 1-based indexing.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

Linear indexing (`LinearIndices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `AbstractVector`): does `v[i]` mean linear indexing , or Cartesian indexing with the array's native indices?

For this reason, your best option may be to iterate over the array with `eachindex(A)`, or, if you require the indices to be sequential integers, to get the index range by calling `LinearIndices(A)`. This will return `axes(A, 1)` if A is an AbstractVector, and the equivalent of `1:length(A)` otherwise.

By this definition, 1-dimensional arrays always use Cartesian indexing with the array's native indices. To help enforce this, it's worth noting that the index conversion functions will throw an error if shape indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than a tuple of `OneTo`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `axes` and `LinearIndices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    axes(dest) == axes(src) || throw(DimensionMismatch("vectors must match"))
```

```
    for i in LinearIndices(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

Allocating storage using generalizations of `similar`

Storage is often allocated with `Array{Int}(undef, dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use. Note that a convenient way of producing an all-zeros array that matches the indices of A is simply `zeros(A)`.

Let's walk through a couple of explicit examples. First, if `A` has conventional indices, then `similar(Array{Int}, axes(A))` would end up calling `Array{Int}(undef, size(A))`, and thus return an array. If `A` is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, axes(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches `A`. (The most obvious implementation is to allocate an `Array{Int}(undef, size(A))` and then "wrap" it in a type that shifts the indices.)

Note also that `similar(Array{Int}, (axes(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of `A`.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Custom `AbstractUnitRange` types

If you're writing a non-1 indexed array type, you will want to specialize `axes` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably not export `ZeroRange` from your package: there may be other packages that implement their own `ZeroRange`, and having multiple distinct `ZeroRange` types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own `ZeroRange` type.

Specializing `axes`

Once you have your `AbstractUnitRange` type, then specialize `axes`:

```
Base.axes(A::ZeroArray) = map(n->ZeroRange(n), A.size)
```

where here we imagine that `ZeroArray` has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `axes(A, d)`:

```
axes(A::AbstractArray{T,N}, d) where {T,N} = d <= N ? axes(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in `Base` there is a dedicated function `indices1` which is equivalent to `axes(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
indices1(A::AbstractArray{T,0}) where {T} = OneTo(1)
indices1(A::AbstractArray) = axes(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing `similar`

Given your custom `ZeroRange` type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end

function Base.similar(f::Union{Function,DataType}, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end
```

Both of these should allocate your custom array type.

Specializing `reshape`

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange,Vararg{ZeroRange}}) = ...
```

and you can `reshape` an array so that the result has custom indices.

For objects that mimic AbstractArray but are not subtypes

`has_offset_axes` depends on having `axes` defined for the objects you call it on. If there is some reason you don't have an `axes` method defined for your object, consider defining a method

```
Base.has_offset_axes(obj::MyNon1IndexedArraylikeObject) = true
```

This will allow code that assumes 1-based indexing to detect a problem and throw a helpful error, rather than returning incorrect results or segfaulting julia.

Catching errors

If your new array type triggers errors in other code, one helpful debugging step can be to comment out `@boundscheck` in your `getindex` and `setindex!` implementation. This will ensure that every element access checks bounds. Or, restart julia with `--check-bounds=yes`.

In some cases it may also be helpful to temporarily disable `size` and `length` for your new array type, since code that makes incorrect assumptions frequently uses these functions.

99.19 Module loading

`Base.require` is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the `import` statement.

Experimental features

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

Module loading callbacks

It is possible to listen to the modules loaded by `Base.require`, by registering a callback.

```
loaded_packages = Channel{Symbol}()
callback = (mod::Symbol) -> put!(loaded_packages, mod)
push!(Base.package_callbacks, callback)
```

Please note that the symbol given to the callback is a non-unique identifier and it is the responsibility of the callback provider to walk the module chain to determine the fully qualified name of the loaded binding.

The callback below is an example of how to do that:

```
# Get the fully-qualified name of a module.
function module_fqn(name::Symbol)
    fqn = fullname(Base.root_module(name))
    return join(fqn, '.')
end
```

99.20 Inference

How inference works

Type inference refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in blog posts (1, 2).

Debugging compiler.jl

You can start a Julia session, edit `compiler/*.jl` (for example to insert `print` statements), and then replace `Core.Compiler` in your running session by navigating to `base/compiler` and executing `include("compiler.jl")`. This trick typically leads to much faster development than if you rebuild Julia for each change.

A convenient entry point into inference is `typeinf_code`. Here's a demo running inference on `convert(Int, UInt(1))`:

```
# Get the method
atypes = Tuple{Type{Int}, UInt}  # argument types
mths = methods(convert, atypes)  # worth checking that there is only one
m = first(mths)

# Create variables needed to call `typeinf_code`
params = Core.Compiler.Params(typemax(UInt))  # parameter is the world age,
                                              #   typemax(UInt) -> most recent
sparams = Core.svec()      # this particular method doesn't have type-parameters
optimize = true            # run all inference optimizations
```

```
cached = false             # force inference to happen (do not use cached results)
Core.Compiler.typeinf_code(m, atypes, sparams, optimize, cached, params)
```

If your debugging adventures require a `MethodInstance`, you can look it up by calling `Core.Compiler.specialize_method` using many of the variables above. A `CodeInfo` object may be obtained with

```
# Returns the CodeInfo object for `convert(Int, ::UInt)`:
ci = (@code_typed convert(Int, UInt(1)))[1]
```

The inlining algorithm (inline_worthy)

Much of the hardest work for inlining runs in `inlining_pass`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `isinlineable` and its primary callee, `inline_worthy`. `isinlineable` handles a number of special cases (e.g., critical functions like `next` and `done`, incorporating a bonus for functions that return tuples, etc.). The main decision-making happens in `inline_worthy`, which returns `true` if the function should be inlined.

`inline_worthy` implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to issue a call to them if they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all `for` loops are analyzed as if they will be executed once, and the cost of an `if...else...end` includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although BaseBenchmarks provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in `add_tfunc` and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on standard ranges for common architectures (see Agner Fog's analysis for more detail).

We supplement this low-level lookup table with a number of special cases. For example, an `:invoke` expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a `:call` expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we assign a cost set by `Params.inline_nonleaf_penalty` (currently set at 1000). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called `statement_cost`. You can run this yourself by following the sketch below, where `f` is your function and `tt` is the Tuple-type of the arguments:

```
# A demo on `fill(3.5, (2, 3))
f = fill
tt = Tuple{Float64, Tuple{Int,Int}}
# Create the objects we need to interact with the compiler
params = Core.Compiler.Params(typemax(UInt))
mi = Base.method_instances(f, tt)[1]
ci = code_typed(f, tt)[1][1]
opt = Core.Compiler.OptimizationState(mi, params)
# Calculate cost of each statement
cost(stmt::Expr) = Core.Compiler.statement_cost(stmt, -1, ci, opt.sptypes, opt.slottypes, opt.params)
cost(stmt) = 0
cst = map(cost, ci.code)

# output

5-element Array{Int64,1}:
  0
  0
 20
 20
  0
```

The output is a `Vector{Int}` holding the estimated cost of each statement in `ci.code`. Note that `ci` includes the consequences of inlining callees, and consequently the costs do too.

99.21 Julia SSA-form IR

Background

Beginning in Julia 0.7, parts of the compiler use a new SSA-form intermediate representation. Historically, the compiler used to directly generate LLVM IR, from a lowered form of the Julia AST. This form had most syntactic abstractions removed, but still looked a lot like an abstract syntax tree. Over time, in order to facilitate optimizations, SSA values were introduced to this IR and the IR was linearized (i.e. a form where function arguments may only be SSA values or constants). However, non-ssa values (slots) remained in the IR due to the lack of Phi nodes in the IR (necessary for back-edges and re-merging of conditional control flow), negating much of the usefulfulness of the SSA form representation to perform middle end optimizations. Some heroic effort was put into making these optimizations work without a complete SSA form representation, but the lack of such a representation ultimately proved prohibitive.

New IR nodes

With the new IR representation, the compiler learned to handle four new IR nodes, Phi nodes, Pi nodes as well as PhiC nodes and Upsilon nodes (the latter two are only used for exception handling).

Phi nodes and Pi nodes

Phi nodes are part of generic SSA abstraction (see the link above if you're not familiar with the concept). In the Julia IR, these nodes are represented as:

```
struct PhiNode
    edges::Vector{Int}
    values::Vector{Any}
end
```

where we ensure that both vectors always have the same length. In the canonical representation (the one handles by codegen and the interpreter), the edge values indicate come-from statement numbers (i.e. if edge has an entry of `15`, there must be a `goto`, `gotoifnot` or implicit fall through from statement `15` that targets this phi node). Values are either SSA values or constants. It is also possible for a value to be unassigned if the variable was not defined on this path. However, undefinedness checks get explicitly inserted and represented as booleans after middle end optimizations, so code generators may assume that any use of a phi node will have an assigned value in the corresponding slot. It is also legal for the mapping to be incomplete, i.e. for a phi node to have missing incoming edges. In that case, it must be dynamically guaranteed that the corresponding value will not be used.

PiNodes encode statically proven information that may be implicitly assumed in basic blocks dominated by a given pi node. They are conceptually equivalent to the technique introduced in the paper "ABCD: Eliminating Array Bounds Checks on Demand" or the predicate info nodes in LLVM. To see how they work, consider, e.g.

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    # use x
else
    # use x
end
```

we can perform predicate insertion and turn this into:

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    %x_int = PiNode(x, Int)
```

```
    # use %x_int
else
    %x_float = PiNode(x, Float64)
    # use %x_float
end
```

Pi nodes are generally ignored in the interpreter, since they don't have any effect on the values, but they may sometimes lead to code generation in the compiler (e.g. to change from an implicitly union split representation to a plain unboxed representation). The main usefulness of PiNodes stems from the fact that path conditions of the values can be accumulated simply by def-use chain walking that is generally done for most optimizations that care about these conditions anyway.

PhiC nodes and Upsilon nodes

Exception handling complicates the SSA story moderately, because exception handling introduces additional control flow edges into the IR across which values must be tracked. One approach to do so, which is followed by LLVM is to make calls which may throw exceptions into basic block terminators and add an explicit control flow edge to the catch handler:

```
invoke @function_that_may_throw() to label %regular unwind to %catch

regular:
# Control flow continues here

catch:
# Exceptions go here
```

However, this is problematic in a language like julia where at the start of the optimization pipeline, we do not know which calls throw. We would have to conservatively assume that every call (which in julia is every statement) throws. This would have several negative effects. On the one hand, it would essentially recuce the scope of every basic block to a single call, defeating the purpose of having operations be performed at the basic block level. On the other hand, every catch basic block would have `n*m` phi node arguments (`n`, the number of statements in the critical region, `m` the number of live values through the catch block). To work around this, we use a combination of `Upsilon` and `PhiC` (the C standing for `catch`, written ϕ^c in the IR pretty printer, because unicode subscript c is not available) nodes. There is several ways to think of these nodes, but perhaps the easiest is to think of each `PhiC` as a load from a unique store-many, read-once slot, with `Upsilon` being the corresponding store operation. The `PhiC` has an operand list of all the upsilon nodes that store to its implicit slot. The `Upsilon` nodes however, do not record which `PhiC` node they store to. This is done for more natural integration with the rest of the SSA IR. E.g. if there are no more uses of a `PhiC` node, it is safe to delete is and the same is true of an `Upsilon` node. In most IR passes, `PhiC` nodes can be treated

similar to `Phi` nodes. One can follow use-def chains through them, and they can be lifted to new `PhiC` nodes and new Upsilon nodes (in the same places as the original `Upsilon` nodes). The result of this scheme is that the number of Upsilon nodes (and `PhiC` arguments) is proportional to the number of assigned values to a particular variable (before SSA conversion), rather than the number of statements in the critical region.

To see this scheme in action, consider the function

```
function foo()
    x = 1
    try
        y = 2
        error()
    catch
    end
    (x, y)
end
```

The corresponding IR (with irrelevant types stripped) is:

```
ir = Code
1 —       nothing
2 —       $(Expr(:enter, 5))
3 — %3  = Y (#undef)|
   %4  = Y (1)|
   %5  = Y (2)|
         Main.bar()|
   %7  = Y (3)└──
       $(Expr(:leave, 1))
4 —       goto 6
5 — %10 = φᶜ (%3, %5)|
   %11 = φᶜ (%4, %7)└──
       $(Expr(:leave, 1))
6 ┄ %13 = φ (4 => 2, 5 => %10)::NotInferenceDontLookHere.MaybeUndef(NotInferenceDontLookHere.Const(2, false))|
   %14 = φ (4 => 3, 5 => %11)::Int64|
         $(Expr(:undefcheck, :y, Core.SSAValue(13)))|
   %16 = Core.tuple(%14, %13)└──
       return %17
```

Note in particular that every value live into the critical region gets an upsilon node at the top of the critical region. This is because catch blocks are considered to have an invisible control flow edge from outside the function. As a result, no SSA value dominates the catch blocks, and all incoming values have to come through a ϕ^c node.

Main SSA data structure

The main `SSAIR` data structure is worthy of discussion. It draws inspiration from LLVM and Webkit's B3 IR. The core of the data structure is a flat vector of statements. Each statement is implicitly assigned an SSA values based on its position in the vector (i.e. the result of the statement at idx 1 can be accessed using `SSAValue(1)` etc). For each SSA value, we additionally maintain its type. Since, SSA values are definitionally assigned only once, this type is also the result type of the expression at the corresponding index. However, while this representation is rather efficient (since the assignments don't need to be explicitly) encoded, if of course carries the drawback that order is semantically significant, so reorderings and insertions change statement numbers. Additionally, we do not keep use lists (i.e. it is impossible to walk from a def to all its uses without explicitly computing this map - def lists however are trivial since you can lookup the corresponding statement from the index), so the LLVM-style RAUW (replace-all-uses-with) operation is unavailable.

Instead, we do the following:

- We keep a separate buffer of nodes to insert (including the position to insert them at, the type of the corresponding value and the node itself). These nodes are numbered by their occurrence in the insertion buffer, allowing their values to be immediately used elesewhere in the IR (i.e. if there is 12 statements in the original statement list, the first new statement will be accessible as `SSAValue(13)`)
- RAUW style operations are performed by setting the corresponding statement index to the replacement value.
- Statements are erased by setting the corresponding statement to `nothing` (this is essentially just a special-case convention of the above
- if there are any uses of the statement being erased they will be set to `nothing`)

There is a `compact!` function that compacts the above data structure by performing the insertion of nodes in the appropriate place, trivial copy propagation and renaming of uses to any changed SSA values. However, the clever part of this scheme is that this compaction can be done lazily as part of the subsequent pass. Most optimization passes need to walk over the entire list of statements, performing analysis or modifications along the way. We provide an `IncrementalCompact` iterator that can be used to iterate over the statement list. It will perform any necessary compaction, and return the new index of the node, as well as the node itself. It is legal at this point to walk def-use chains, as well as make any modifications or deletions to the IR (insertions are disallowed however).

The idea behind this arrangement is that, since the optimization passes need to touch the corresponding memory anyway, and incur the corresponding memory access penalty, performing the extra housekeeping should have comparatively little overhead (and save the overhead of maintaining these data structures during IR modification).

99.22 Static analyzer annotations for GC correctness in C code

Running the analysis

The analyzer plugin that drives the anlysis ships with julia. Its source code can be found in `src/clangsa`. Running it requires the clang dependency to be build. Set the `BUILD_LLVM_CLANG` variable in your Make.user in order to build an appropriate version of clang. You may also want to use the prebuilt binaries using the `USE_BINARYBUILDER_LLVM` options. Afterwards, running the analysis over the source tree is as simple as running `make -C src analyzegc`.

General Overview

Since Julia's GC is precise, it needs to maintain correct rooting information for any value that may be referenced at any time GC may occur. These places are known as `safepoints` and in the function local context, we extend this designation to any function call that may recursively end up at a safepoint.

In generated code, this is taken care of automatically by the GC root placement pass (see the chapter on GC rooting in the LLVM codegen devdocs). However, in C code, we need to inform the runtime of any GC roots manually. This is done using the following macros:

```
// The value assigned to any slot passed as an argument to these
// is rooted for the duration of this GC frame.
JL_GC_PUSH{1,...,6}(args...)
// The values assigned into the size `n` array `rts` are rooted
// for the duration of this GC frame.
JL_GC_PUSHARGS(rts, n)
// Pop a GC frame
JL_GC_POP
```

If these macros are not used where they need to be, or they are used incorrectly, the result is silent memory corruption. As such it is very important that they are placed correctly in all applicable code.

As such, we employ static analysis (and in particular the clang static analyzer) to help ensure that these macros are used correctly. The remainder of this document gives an overview of this static analysis and describes the support needed in the julia code base to make things work.

GC Invariants

There is two simple invariants correctness:

- All `GC_PUSH` calls need to be followed by an appropriate `GC_POP` (in practice we enforce this at the function level)

- If a value was previously not rooted at any safepoint, it may no longer be referenced afterwards

Of course the devil is in the details here. In particular to satisfy the second of the above conditions, we need to know:

- Which calls are safepoints and which are not
- Which values are rooted at any given safepoint and which are not
- When is a value referenced

For the second point in particular, we need to know which memory locations will be considered rooting at runtime (i.e. values assigned to such locations are rooted). This includes locations explicitly designated as such by passing them to one of the `GC_PUSH` macros, globally rooted locations and values, as well as any location recursively reachable from one of those locations.

Static Analysis Algorithm

The idea itself is very simple, although the implementation is quite a bit more complicated (mainly due to a large number of special cases and intricacies of C and C++). In essence, we keep track of all locations that are rooting, all values that are rootable and any expression (assignments, allocations, etc) affect the rootedness of any rootable values. Then, at any safepoint, we perform a "symbolic GC" and poison any values that are not rooted at said location. If these values are later referenced, we emit an error.

The clang static analyzer works by constructing a graph of states and exploring this graph for sources of errors. Several nodes in this graph are generated by the analyzer itself (e.g. for control flow), but the definitions above augment this graph with our own state.

The static analyzer is interprocedural and can analyze control flow across function boundaries. However, the static analyzer is not fully recursive and makes heuristic decisions about which calls to explore (additionally some calls are cross-translation unit and invisible to the analyzer). In our case, our definition of correctness requires total information. As such, we need to annotate the prototypes of all function calls with whatever information the analysis required, even if that information would otherwise be available by interprocedural static analysis.

Luckily however, we can still use this interprocedural analysis to ensure that the annotations we place on a given function are indeed correct given the implementation of said function.

The analyzer annotations

These annotations are found in src/support/analyzer_annotations.h. The are only active when the analyzer is being used and expand either to nothing (for prototype annotations) or to no-ops (for function like annotations).

`JL_NOTSAFEPOINT`

This is perhaps the most common annotation, and should be placed on any function that is known not to possibly lead to reaching a GC safepoint. In general, it is only safe for such a function to perform arithmetic, memory accesses and calls to functions either annotated `JL_NOTSAFEPOINT` or otherwise known not to be safepoints (e.g. function in the C standard library, which are hardcoded as such in the analyzer)

It is valid to keep values unrooted across calls to any function annotated with this attribute:

Usage Example:

```
void jl_get_one() JL_NOTSAFEPOINT {
  return 1;
}

jl_value_t *example() {
  jl_value_t *val = jl_alloc_whatever();
  // This is valid, even though `val` is unrooted, because
  // jl_get_one is not a safepoint
  jl_get_one();
  return val;
}
```

`JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY`

When `JL_MAYBE_UNROOTED` is annotated as an argument on a function, indicates that said argument may be passed, even if it is not rooted. In the ordinary course of events, the julia ABI guarantees that callers root values before passing them to callees. However, some functions do not follow this ABI and allow values to be passed to them even though they are not rooted. Note however, that this does not automatically imply that said argument will be preserved. The `ROOTS_TEMPORARILY` annotation provides the stronger guarantee that, not only may the value be unrooted when passed, it will also be preserved across any internal safepoints by the callee.

Note that `JL_NOTSAFEPOINT` essentially implies `JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY`, because the rootedness of an argument is irrelevant if the function contains no safepoints.

One additional point to note is that these annotations apply on both the caller and the callee side. On the caller side, they lift rootedness restrictions that are normally required for julia ABI functions. On the callee side, they have the reverse effect of preventing these arguments from being considered implicitly rooted.

If either of these annotations is applied to the function as a whole, it applies to all arguments of the function. This should generally only be necessary for varargs functions.

Usage example:

```
JL_DLLEXPORT void JL_NORETURN jl_throw(jl_value_t *e JL_MAYBE_UNROOTED);
jl_value_t *jl_alloc_error();

void example() {
  // The return value of the allocation is unrooted. This would normally
  // be an error, but is allowed because of the above annotation.
  jl_throw(jl_alloc_error());
}
```

JL_PROPAGATES_ROOT

This annotation is commonly found on accessor functions that return one rootable object stored within another. When annotated on a function argument, it tells the analyzer that the root for that argument also applies to the value returned by the function.

Usage Example:

```
jl_value_t *jl_svecref(jl_svec_t *t JL_PROPAGATES_ROOT, size_t i) JL_NOTSAFEPOINT;

size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_svecref(svec, 1)
  // This is valid, because, as annotated by the PROPAGATES_ROOT annotation,
  // jl_svecref propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_ROOTING_ARGUMENT/JL_ROOTED_ARGUMENT

This is essentially the assignment counterpart to `JL_PROPAGATES_ROOT`. When assigning a value to a field of another value that is already rooted, the assigned value will inherit the root of the value it is assigned into.

Usage Example:

```
void jl_svecset(void *t JL_ROOTING_ARGUMENT, size_t i, void *x JL_ROOTED_ARGUMENT) JL_NOTSAFEPOINT



size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_box_long(10000);
  jl_svecset(svec, val);
  // This is valid, because the annotations imply that the
  // jl_svecset propagates the rooted-ness from `svec` to `val`
```

```
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

`JL_GC_DISABLED`

This annotation implies that this function is only called with the GC runtime-disabled. Functions of this kind are most often encountered during startup and in the GC code itself. Note that this annotation is checked against the runtime enable/disable calls, so clang will know if you lie. This is not a good way to disable processing of a given function if the GC is not actually disabled (use `ifdef __clang_analyzer__` for that if you must).

Usage example:

```
void jl_do_magic() JL_GC_DISABLED {
  // Wildly allocate here with no regard for roots
}

void example() {
  int en = jl_gc_enable(0);
  jl_do_magic();
  jl_gc_enable(en);
}
```

`JL_REQUIRE_ROOTED_SLOT`

This annotation requires the caller to pass in a slot that is rooted (i.e. values assigned to this slot will be rooted).

Usage example:

```
void jl_do_processing(jl_value_t **slot JL_REQUIRE_ROOTED_SLOT) {
  *slot = jl_box_long(1);
  // Ok, only, because the slot was annotated as rooting
  jl_gc_safepoint();
}

void example() {
  jl_value_t *slot = NULL;
  JL_GC_PUSH1(&slot);
  jl_do_processing(&slot);
  JL_GC_POP();
}
```

JL_GLOBALLY_ROOTED

This annotation implies that a given value is always globally rooted. It can be applied to global variable declarations, in which case it will apply to the value of those variables (or values if the declaration if for an array), or to functions, in which case it will apply to the return value of such functions (e.g. for functions that always return some private, globally rooted value).

Usage example:

```
extern JL_DLLEXPORT jl_datatype_t *jl_any_type JL_GLOBALLY_ROOTED;
jl_ast_context_t *jl_ast_ctx(fl_context_t *fl) JL_GLOBALLY_ROOTED;
```

JL_ALWAYS_LEAFTYPE

This annotations is essentially equivalent to `JL_GLOBALLY_ROOTED`, except that is should only be used if those values are globally rooted by virtue of being a leaftype. The rooting of leaftypes is a bit complicated. They are generally rooted through `cache` field of the corresponding `TypeName`, which itself is rooted by the containing module (so they're rooted as long as the containing module is ok) and we can generally assume that leaftypes are rooted where they are used, but we may refine this property in the future, so the separate annotation helps split out the reason for being globally rooted.

The analyzer also automatically detects checks for leaftype-ness and will not complain about missing GC roots on these paths.

```
JL_DLLEXPORT jl_value_t *jl_apply_array_type(jl_value_t *type, size_t dim) JL_ALWAYS_LEAFTYPE;
```

JL_GC_PROMISE_ROOTED

This is a function-like annotation. Any value passed to this annotation will be considered rooted for the scope of the current function. It is designed as an escape hatch for analyzer inadequacy or complicated situations. However, it should be used sparingly, in favor of improving the analyzer itself.

```
void example() {
  jl_value_t *val = jl_alloc_something();
  if (some_condition) {
    // We happen to know for complicated external reasons
    // that val is rooted under these conditions
    JL_GC_PROMISE_ROOTED(val);
  }
}
```

Completeness of analysis

The analyzer only looks at local information. In particular, e.g. in the `PROPAGATES_ROOT` case above, it assumes that such memory is only modified in ways it can see, not in any called functions (unless it happens to decide to consider them in its analysis) and not in any concurrently running threads. As such, it may miss a few problematic cases, though in practice such concurrent modification is fairly rare. Improving the analyzer to handle more such cases may be an interesting topic for future work.

Stack frame	Source code	Notes
jl_uv_write()	jl_uv.c	called though ccall
julia_write_282942	stream.jl	function write!(s::IO, a::Array{T}) where T
julia_print_284639	ascii.jl	print(io::IO, s::String) = (write(io, s); nothing)
jlcall_print_284639		
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String, Char, Char...)
jl_apply()	julia.h	
jl_f_apply()	builtins.c	
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(Base.TTY, String, String...)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(String,)
jl_apply()	julia.h	
do_call()	interpreter.c	
eval()	interpreter.c	
jl_interpret_toplevel_expr()	interpreter.c	
jl_toplevel_eval_flex()	toplevel.c	
jl_toplevel_eval()	toplevel.c	
jl_toplevel_eval_in()	builtins.c	
jl_f_top_eval()	builtins.c	

Input	AST
`f(x)`	`(call f x)`
`f(x, y=1, z=2)`	`(call f x (kw y 1) (kw z 2))`
`f(x; y=1)`	`(call f (parameters (kw y 1)) x)`
`f(x...)`	`(call f (... x))`

Input	AST
`x+y`	`(call + x y)`
`a+b+c+d`	`(call + a b c d)`
`2x`	`(call * 2 x)`
`a&&b`	`(&& a b)`
`x += 1`	`(+= x 1)`
`a ? 1 : 2`	`(if a 1 2)`
`a:b`	`(: a b)`
`a:b:c`	`(: a b c)`
`a,b`	`(tuple a b)`
`a==b`	`(call == a b)`
`1<i<=n`	`(comparison 1 < i <= n)`
`a.b`	`(. a (quote b))`
`a.(b)`	`(. a b)`

Input	AST
`a[i]`	`(ref a i)`
`t[i;j]`	`(typed_vcat t i j)`
`t[i j]`	`(typed_hcat t i j)`
`t[a b; c d]`	`(typed_vcat t (row a b) (row c d))`
`a{b}`	`(curly a b)`
`a{b;c}`	`(curly a (parameters c) b)`
`[x]`	`(vect x)`
`[x,y]`	`(vect x y)`
`[x;y]`	`(vcat x y)`
`[x y]`	`(hcat x y)`
`[x y; z t]`	`(vcat (row x y) (row z t))`
`[x for y in z, a in b]`	`(comprehension x (= y z) (= a b))`
`T[x for y in z]`	`(typed_comprehension T x (= y z))`
`(a, b, c)`	`(tuple a b c)`
`(a; b; c)`	`(block a (block b c))`

Input	AST
`@m x y`	`(macrocall @m (line) x y)`
`Base.@m x y`	`(macrocall (. Base (quote @m)) (line) x y)`
`@Base.m x y`	`(macrocall (. Base (quote @m)) (line) x y)`

Input	AST
`"a"`	`"a"`
`x"y"`	`(macrocall @x_str (line) "y")`
`x"y"z`	`(macrocall @x_str (line) "y" "z")`
`"x = $x"`	`(string "x = " x)`
`` `a b c` ``	`(macrocall @cmd (line) "a b c")`

Input	AST
`import a`	`(import (. a))`
`import a.b.c`	`(import (. a b c))`
`import ...a`	`(import (. . . . a))`
`import a.b, c.d`	`(import (. a b) (. c d))`
`import Base: x`	`(import (: (. Base) (. x)))`
`import Base: x, y`	`(import (: (. Base) (. x) (. y)))`
`export a, b`	`(export a b)`

Input	AST
`11111111111111111111`	`(macrocall @int128_str (null) "11111111111111111111")`
`0xffffffffffffffff`	`(macrocall @uint128_str (null) "0xffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str (null) "1111....")`

Name	Prefix	Purpose
Native	`julia_`	Speed via specialized signatures
JL Call	`jlcall_`	Wrapper for generic calls
JL Call	`jl_`	Builtins
C ABI	`jlcapi_`	Wrapper callable from C

File	Description
`builtins.c`	Builtin functions
`ccall.cpp`	Lowering `ccall`
`cgutils.cpp`	Lowering utilities, notably for array and tuple accesses
`codegen.cpp`	Top-level of code generation, pass list, lowering builtins
`debuginfo.cpp`	Tracks debug information for JIT code
`disasm.cpp`	Handles native object file and JIT code diassembly
`gf.c`	Generic functions
`intrinsics.cpp`	Lowering intrinsics
`llvm-simdloop.cpp`	Custom LLVM pass for `@simd`
`sys.c`	I/O and operating system utility functions

Chapter 100

Developing/debugging Julia's C code

100.1 Reporting and analyzing crashes (segfaults)

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

- Segfaults during bootstrap (`sysimg.jl`)
- Segfaults when running a script
- Errors during Julia startup

Version/Environment info

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. Please also include the output of `versioninfo()` (exported from the `InteractiveUtils` standard library) in any report you create:

```
julia> using InteractiveUtils

julia> versioninfo()
Julia Version 1.5.3
Commit 788b2c77c1 (2020-11-09 13:37 UTC)
Platform Info:
```

```
  OS: macOS (x86_64-apple-darwin18.7.0)
  CPU: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-9.0.1 (ORCJIT, skylake)
Environment:
  JULIA_CUDA_SILENT = 1
```

Segfaults during bootstrap (`sysimg.jl`)

Segfaults toward the end of the `make` process of building Julia are a common symptom of something going wrong while Julia is preparsing the corpus of code in the `base/` folder. Many factors can contribute toward this process dying unexpectedly, however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of `gdb`. Explicitly:

Create a debug build of Julia:

```
$ cd <julia_root>
$ make debug
```

Note that this process will likely fail with the same error as a normal `make` incantation, however this will create a debug executable that will offer `gdb` the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of `gdb`:

```
$ cd base/
$ gdb -x ../contrib/debug_bootstrap.gdb
```

This will start `gdb`, attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit `<enter>` a few times to get the full backtrace. Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Segfaults when running a script

The procedure is very similar to Segfaults during bootstrap (`sysimg.jl`). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
$ cd <julia_root>
$ make debug
$ gdb --args usr/bin/julia-debug <path_to_your_script>
```

Note that `gdb` will sit there, waiting for instructions. Type `r` to run the process, and `bt` to generate a backtrace once it segfaults:

```
(gdb) r
Starting program: /home/sabae/src/julia/usr/bin/julia-debug ./test.jl
...
(gdb) bt
```

Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

```
$ julia
exec: error -5
```

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the `julia` process:

- On Linux, use `strace`:

  ```
  $ strace julia
  ```

- On OSX, use `dtruss`:

  ```
  $ dtruss -f julia
  ```

Create a gist with the `strace`/ `dtruss` output, the version info, and any other pertinent information and open a new issue on Github with a link to the gist.

Glossary

A few terms have been used as shorthand in this guide:

- `<julia_root>` refers to the root directory of the Julia source tree; e.g. it should contain folders such as `base`, `deps`, `src`, `test`, etc.....

100.2 gdb debugging tips

Displaying Julia variables

Within `gdb`, any `jl_value_t*` object `obj` can be displayed using

```
(gdb) call jl_(obj)
```

The object will be displayed in the `julia` session, not in the gdb session. This is a useful way to discover the types and values of objects being manipulated by Julia's C code.

Similarly, if you're debugging some of Julia's internals (e.g., `compiler.jl`), you can print `obj` using

```
ccall(:jl_, Cvoid, (Any,), obj)
```

This is a good way to circumvent problems that arise from the order in which julia's output streams are initialized.

Julia's flisp interpreter uses `value_t` objects; these can be displayed with `call fl_print(fl_ctx, ios_stdout, obj)`.

Useful Julia variables for Inspecting

While the addresses of many variables, like singletons, can be be useful to print for many failures, there are a number of additional variables (see `julia.h` for a complete list) that are even more useful.

- (when in `jl_apply_generic`) `mfunc` and `jl_uncompress_ast(mfunc->def, mfunc->code)` :: for figuring out a bit about the call-stack
- `jl_lineno` and `jl_filename` :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- `$1` :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as `print`)
- `jl_options` :: sometimes useful, since it lists all of the command line options that were successfully parsed
- `jl_uv_stderr` :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- `jl_gdblookup($rip)` :: For looking up the current function and line. (use `$eip` on i686 platforms)
- `jlbacktrace()` :: For dumping the current Julia backtrace stack to stderr. Only usable after `record_backtrace()` has been called.

- `jl_dump_llvm_value(Value*)` :: For invoking `Value->dump()` in gdb, where it doesn't work natively. For example, `f->linfo->functionObject`, `f->linfo->specFunctionObject`, and `to_function(f->linfo)`.
- `Type->dump()` :: only works in lldb. Note: add something like `;1` to prevent lldb from printing its prompt over the output
- `jl_eval_string("expr")` :: for invoking side-effects to modify the current state or to lookup symbols
- `jl_typeof(jl_value_t*)` :: for extracting the type tag of a Julia value (in gdb, call `macro define jl_typeof jl_typeof` first, or pick something short like `ty` for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your `gdb` session, set a breakpoint in `jl_breakpoint` like so:

```
(gdb) break jl_breakpoint
```

Then within your Julia code, insert a call to `jl_breakpoint` by adding

```
ccall(:jl_breakpoint, Cvoid, (Any,), obj)
```

where `obj` can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the `jl_apply` frame, from which you can display the arguments to a function using, e.g.,

```
(gdb) call jl_(args[0])
```

Another useful frame is `to_function(jl_method_instance_t *li, bool cstyle)`. The `jl_method_instance_t*` argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call `jl_uncompress_ast` and then pass the result to `jl_`:

```
#2  0x00007ffff7928bf7 in to_function (li=0x2812060, cstyle=false) at codegen.cpp:584
584         abort();
(gdb) p jl_(jl_uncompress_ast(li, li->ast))
```

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is `sysimg.jl`:

```
(gdb) break jl_load if strcmp(fname, "sysimg.jl")==0
```

Calling a particular method

```
(gdb) break jl_apply_generic if strcmp((char*)(jl_symbol_name)(jl_gf_mtable(F)->name), "method_to_break")==0
```

Since this function is used for every call, you will make everything 1000x slower if you do this.

Dealing with signals

Julia requires a few signal to function property. The profiler uses `SIGUSR2` for sampling and the garbage collector uses `SIGSEGV` for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace `SIGSEGV` with `SIGUSRS` or other signals you want to ignore):

```
(gdb) handle SIGSEGV noprint nostop pass
```

The corresponding LLDB command is (after the process is started):

```
(lldb) pro hand -p true -s false -n false SIGSEGV
```

If you are debugging a segfault with threaded code, you can set a breakpoint on `jl_critical_error` (`sigdie_handler` should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

Debugging during Julia's build process (bootstrap)

Errors that occur during `make` need special handling. Julia is built in two stages, constructing `sys0` and `sys.ji`. To see what commands are running at the time of failure, use `make VERBOSE=1`.

At the time of this writing, you can debug build errors during the `sys0` phase from the `base` directory using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys0 sysimg.jl
```

You might need to delete all the files in `usr/lib/julia/` to get this to work.

You can debug the `sys.ji` phase using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys -J ../usr/lib/julia/sys0.ji
     sysimg.jl
```

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in `jl_error` (there are several other useful spots, for specific kinds of failures, including: `jl_too_few_args`, `jl_too_many_args`, and `jl_throw`).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to `jl_apply`. To take a real-world example:

```
Breakpoint 1, jl_throw (e=0x7ffdf42de400) at task.c:802
802 {
(gdb) p jl_(e)
ErrorException("auto_unbox: unable to determine argument type")
$2 = void
(gdb) bt 10
#0  jl_throw (e=0x7ffdf42de400) at task.c:802
#1  0x00007ffff65412fe in jl_error (str=0x7ffde56be000 <_j_str267> "auto_unbox:
   unable to determine argument type")
   at builtins.c:39
#2  0x00007ffde56bd01a in julia_convert_16886 ()
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h:1281
...
```

The most recent `jl_apply` is at frame #3, so we can go back there and look at the AST for the function `julia_convert_16886`. This is the uniqued name for some method of `convert`. `f` in this frame is a `jl_function_t*`, so we can look at the type signature, if any, from the `specTypes` field:

```
(gdb) f 3
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h:1281
1281            return f->fptr((jl_value_t*)f, args, nargs);
(gdb) p f->linfo->specTypes
$4 = (jl_tupletype_t *) 0x7ffdf39b1030
(gdb) p jl_( f->linfo->specTypes )
Tuple{Type{Float32}, Float64}           # <-- type signature for julia_convert_16886
```

Then, we can look at the AST for this function:

```
(gdb) p jl_( jl_uncompress_ast(f->linfo, f->linfo->ast) )
Expr(:lambda, Array{Any, 1}[:#s29, :x], Array{Any, 1}[Array{Any, 1}[], Array{Any, 1}[Array{Any, 1}[:#s29, :Any, 0],
     Array{Any, 1}[:x, :Any, 0]], Array{Any, 1}[], 0], Expr(:body,
Expr(:line, 90, :float.jl)::Any,
Expr(:return, Expr(:call, :box, :Float32, Expr(:call, :fptrunc, :Float32, :x)::Any)::Any)::Any)::Any)::Any
```

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached `functionObject` from the `jl_lamdbda_info_t*`:

```
(gdb) p f->linfo->functionObject
$8 = (void *) 0x1289d070
(gdb) set f->linfo->functionObject = NULL
```

Then, set a breakpoint somewhere useful (e.g. `emit_function`, `emit_expr`, `emit_call`, etc.), and run codegen:

```
(gdb) p jl_compile(f)
... # your breakpoint here
```

Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

```
(gdb) attach -w -n julia-debug
```

or:

```
(lldb) process attach -w -n julia-debug
```

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each `fork` from the parent process)

Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with rr, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of rr (3.1.0 or higher) is required.

100.3 Using Valgrind with Julia

Valgrind is a tool for memory debugging, memory leak detection, and profiling. This section describes things to keep in mind when using Valgrind to debug memory issues with Julia.

General considerations

By default, Valgrind assumes that there is no self modifying code in the programs it runs. This assumption works fine in most instances but fails miserably for a just-in-time compiler like `julia`. For this reason it is crucial to pass `--smc-check=all-non-file` to `valgrind`, else code may crash or behave unexpectedly (often in subtle ways).

In some cases, to better detect memory errors using Valgrind it can help to compile `julia` with memory pools disabled. The compile-time flag `MEMDEBUG` disables memory pools in Julia, and `MEMDEBUG2` disables memory pools in FemtoLisp. To build `julia` with both flags, add the following line to `Make.user`:

```
CFLAGS = -DMEMDEBUG -DMEMDEBUG2
```

Another thing to note: if your program uses multiple workers processes, it is likely that you want all such worker processes to run under Valgrind, not just the parent process. To do this, pass `--trace-children=yes` to `valgrind`.

Suppressions

Valgrind will typically display spurious warnings as it runs. To reduce the number of such warnings, it helps to provide a suppressions file to Valgrind. A sample suppressions file is included in the Julia source distribution at `contrib/valgrind-julia.supp`.

The suppressions file can be used from the `julia/` source directory as follows:

```
$ valgrind --smc-check=all-non-file --suppressions=contrib/valgrind-julia.supp ./julia progname.jl
```

Any memory errors that are displayed should either be reported as bugs or contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the `julia/test/` directory:

```
valgrind --smc-check=all-non-file --trace-children=yes --suppressions=$PWD/../contrib/valgrind-julia.supp ../julia
     runtests.jl all
```

If you would like to see a report of "definite" memory leaks, pass the flags `--leak-check=full --show-leak-kinds=definite` to `valgrind` as well.

Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting `--smc-check=all-non-file` you find that your program behaves differently when run under Valgrind, it may help to pass `--tool=none` to `valgrind` as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

100.4 Sanitizer support

General considerations

Using Clang's sanitizers obviously require you to use Clang (`USECLANG=1`), but there's another catch: most sanitizers require a run-time library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies on functionality from that library. This implies that the LLVM version of your host compiler matches that of the LLVM library used within Julia.

An easy solution is to have an dedicated build folder for providing a matching toolchain, by building with `BUILD_LLVM_CLANG=1`. You can then refer to this toolchain from another build folder by specifying `USECLANG=1` while overriding the `CC` and `CXX` variables.

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with `SANITIZE=1` you enable ASAN for the Julia compiler and its generated code. In addition, you can specify `LLVM_SANITIZE=1` to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes `testall1` takes 8-10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the `allow_user_segv_handler=1` ASAN flag, which is required for signal delivery to work properly. You can define other options using the `ASAN_OPTIONS` environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying `fast_unwind_on_malloc=0` and `malloc_context_size=2`, at the cost of backtrace accuracy. For now, Julia also sets `detect_leaks=0`, but this should be removed in the future.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with `SANITIZE_MEMORY=1`.

Part V

Julia v1.3 Release Notes

Chapter 101

New language features

- Support for Unicode 12.1.0 (#32002).
- Methods can now be added to an abstract type (#31916).

Chapter 102

Language changes

Chapter 103

Multi-threading changes

- All system-level I/O operations (e.g. files and sockets) are now thread-safe. This does not include subtypes of `IO` that are entirely in-memory, such as `IOBuffer`, although it specifically does include `BufferStream`. (#32309, #32174, #31981, #32421).
- The global random number generator (`GLOBAL_RNG`) is now thread-safe (and thread-local) (#32407).
- New experimental `Threads.@spawn` macro that runs a task on any available thread (#32600).

Chapter 104

Build system changes

Chapter 105

New library functions

- `findfirst`, `findlast`, `findnext` and `findprev` now accept a character as first argument to search for that character in a string passed as the second argument (#31664).
- New `findall(pattern, string)` method where `pattern` is a string or regex (#31834).
- `istaskfailed` is now documented and exported, like its siblings `istaskdone` and `istaskstarted` (#32300).
- `RefArray` and `RefValue` objects now accept index `CartesianIndex()` in `getindex` and `setindex!` (#32653)

Chapter 106

Standard library changes

- `Regex` can now be multiplied (`*`) and exponentiated (`^`), like strings (#23422).
- `Cmd` interpolation (`` `$(x::Cmd) a b c` `` where) now propagates `x`'s process flags (environment, flags, working directory, etc) if `x` is the first interpolant and errors otherwise (#24353).
- Zero-dimensional arrays are now consistently preserved in the return values of mathematical functions that operate on the array(s) as a whole (and are not explicitly broadcasted across their elements). Previously, the functions `+`, `-`, `*`, `/`, `conj`, `real` and `imag` returned the unwrapped element when operating over zero-dimensional arrays (#32122).
- `IPAddr` subtypes now behave like scalars when used in broadcasting (#32133).
- `clamp` can now handle missing values (#31066).
- `empty` now accepts a `NamedTuple` (#32534).
- `mod` now accepts a unit range as the second argument to easily perform offset modular arithmetic to ensure the result is inside the range (#32628).

Libdl

- `dlopen()` can now be invoked in `do`-block syntax, similar to `open()`.

LinearAlgebra

- The BLAS submodule no longer exports `dot`, which conflicts with that in LinearAlgebra (#31838).
- `diagm` and `spdiagm` now accept optional `m,n` initial arguments to specify a size (#31654).

- `Hessenberg` factorizations `H` now support efficient shifted solves `(H+μI) \ b` and determinants, and use a specialized tridiagonal factorization for Hermitian matrices. There is also a new `UpperHessenberg` matrix type (#31853).

SparseArrays

- `SparseMatrixCSC(m,n,colptr,rowval,nzval)` perform consistency checks for arguments: `colptr` must be properly populated and lengths of `colptr`, `rowval`, and `nzval` must be compatible with `m`, `n`, and `eltype(colptr)`.
- `sparse(I, J, V, m, n)` verifies lengths of `I`, `J`, `V` are equal and compatible with `eltype(I)` and `m`, `n`.
- The `sprand` function is now 2 to 5 times faster (#30494). As a consequence of this change, the random stream of matrices produced with `sprand` and `sprandn` has changed.

Dates

- `DateTime` and `Time` formatting/parsing now supports 12-hour clocks with AM/PM via `I` and `p` codes, similar to `strftime` (#32308).
- Fixed `repr` such that it displays `Time` as it would be entered in Julia (#32103).

Sockets

- `getipaddrs` returns IP addresses in the order provided by libuv (#32260).
- `getipaddr` prefers to return the first `IPv4` interface address provided by libuv (#32260).

Statistics

- `mean` now accepts both a function argument and a `dims` keyword (#31576).

Sockets

- Added `InetAddr` constructor from `AbstractString`, representing IP address, and `Integer`, representing port number (#31459).

Miscellaneous

- `foldr` and `mapfoldr` now work on any iterator that supports `Iterators.reverse`, not just arrays (#31781).

Chapter 107

Deprecated or removed

- `@spawn expr` from the `Distributed` standard library should be replaced with `@spawnat :any expr` (#32600).

Chapter 108

External dependencies

Chapter 109

Tooling Improvements

- The `ClangSA.jl` static analysis package has been imported, which makes use of the clang static analyzer to validate GC invariants in Julia's C code. The analysis may be run using `make -C src analyzegc`.