The Julia Language

The Julia Project

August 12, 2018

Contents

Part I

Home

Part II

Julia 1.0 Documentation

Welcome to the documentation for Julia 1.0.

Please read the release blog post for a general overview of the language and many of the changes since Julia v0.6. Note that version 0.7 was released alongside 1.0 to provide an upgrade path for packages and code that predates the 1.0 release. The only difference between 0.7 and 1.0 is the removal of deprecation warnings. For a complete list of all the changes since 0.6, see the release notes for version 0.7

0.1 Introduction

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. We believe there are many good reasons to prefer dynamic languages for these applications, and we do not expect their use to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. If you find that something is slow, we highly recommend reading through the Performance Tips section before trying anything else. Once you understand how Julia works, it's easy to write code that's nearly as fast as C.

Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation, implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby.

The most significant departures of Julia from typical dynamic languages are:

- The core language imposes very little; Julia Base and the standard library is written in Julia itself, including primitive operations like integer arithmetic

- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations

- The ability to define function behavior across many combinations of argument types via multiple dispatch

- Automatic generation of efficient, specialized code for different argument types

- Good performance, approaching that of statically-compiled languages like C

Although one sometimes speaks of dynamic languages as being "typeless", they are definitely not: every object, whether primitive or user-defined, has a type. The lack of type declarations in most dynamic languages, however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. In static languages, on the other hand, while one can – and usually must – annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler.

While the casual programmer need not explicitly use types or multiple dispatch, they are the core unifying features of Julia: functions are defined on different combinations of argument types, and applied by dispatching to the most specific matching definition. This model is a good fit for mathematical programming, where it is unnatural for the first argument to "own" an operation as in traditional object-oriented dispatch. Operators are just functions with special

notation – to extend addition to new user-defined data types, you define new methods for the + function. Existing code then seamlessly applies to the new data types.

Partly because of run-time type inference (augmented by optional type annotations), and partly because of a strong focus on performance from the inception of the project, Julia's computational efficiency exceeds that of other dynamic languages, and even rivals that of statically-compiled languages. For large scale numerical problems, speed always has been, continues to be, and probably always will be crucial: the amount of data being processed has easily kept pace with Moore's Law over the past decades.

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- Free and open source (MIT licensed)

- User-defined types are as fast and compact as built-ins

- No need to vectorize code for performance; devectorized code is fast

- Designed for parallelism and distributed computation

- Lightweight "green" threading (coroutines)

- Unobtrusive yet powerful type system

- Elegant and extensible conversions and promotions for numeric and other types

- Efficient support for Unicode, including but not limited to UTF-8

- Call C functions directly (no wrappers or special APIs needed)

- Powerful shell-like capabilities for managing other processes

- Lisp-like macros and other metaprogramming facilities

Part III

Manual

Chapter 1

Getting Started

Julia installation is straightforward, whether using precompiled binaries or compiling from source. Download and install Julia by following the instructions at https://julialang.org/downloads/.

The easiest way to learn and experiment with Julia is by starting an interactive session (also known as a read-eval-print loop or "REPL") by double-clicking the Julia executable or running `julia` from the command line:

```
$ julia

               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.0.0 (2018-08-11)
 _/ |\__'_|_|_|\__'_|  |  Commit e7d15d4a01* (0 days old master)
|__/                   |


julia> 1 + 2
3

julia> ans
3
```

To exit the interactive session, type CTRL-D (press the Control/^ key together with the d key), or type `exit()`. When run in interactive mode, `julia` displays a banner and prompts the user for input. Once the user has entered a complete expression, such as `1 + 2`, and hits enter, the interactive session evaluates the expression and shows its value. If an expression is entered into an interactive session with a trailing semicolon, its value is not shown. The variable ans is bound to the value of the last evaluated expression whether it is shown or not. The ans variable is only bound in interactive sessions, not when Julia code is run in other ways.

To evaluate expressions written in a source file `file.jl`, write `include("file.jl")`.

To run code in a file non-interactively, you can give it as the first argument to the `julia` command:

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

As the example implies, the following command-line arguments to `julia` are interpreted as command-line arguments to the program `script.jl`, passed in the global constant ARGS. The name of the script itself is passed in as the global PROGRAM_FILE. Note that ARGS is also set when a Julia expression is given using the -e option on the command line (see the `julia` help output below) but PROGRAM_FILE will be empty. For example, to just print the arguments given to a script, you could do this:

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

foo
bar
```

Or you could put that code into a script and run it:

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

The -- delimiter can be used to separate command-line arguments intended for the script file from arguments intended for Julia:

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

Julia can be started in parallel mode with either the -p or the --machine-file options. -p n will launch an additional n worker processes, while --machine-file file will launch a worker for each line in file file. The machines defined in file must be accessible via a password-less ssh login, with Julia installed at the same location as the current host. 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.

If you have code that you want executed whenever Julia is run, you can put it in ~/.julia/config/startup.jl:

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

...
```

There are various ways to run Julia code and provide options, similar to those available for the perl and ruby programs:

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

1.1   Resources

A curated list of useful learning resources to help new users get started can be found on the learning page of the main Julia web site.

Switch	Description
`-v, --version`	Display version information
`-h, --help`	Print this message
`-J, --sysimage <file>`	Start up with the given system image file
`-H, --home <dir>`	Set location of `julia` executable
`--startup-file={yes|no}`	Load `~/.julia/config/startup.jl`
`--handle-signals={yes|no}`	Enable or disable Julia's default signal handlers
`--sysimage-native-code={yes|no}`	Use native code from system image if available
`--compiled-modules={yes|no}`	Enable or disable incremental precompilation of modules
`-e, --eval <expr>`	Evaluate `<expr>`
`-E, --print <expr>`	Evaluate `<expr>` and display the result
`-L, --load <file>`	Load `<file>` immediately on all processors
`-p, --procs {N|auto}`	Integer value N launches N additional local worker processes; `auto` launches as many workers as the number of local CPU threads (logical cores)
`--machine-file <file>`	Run processes on hosts listed in `<file>`
`-i`	Interactive mode; REPL runs and `isinteractive()` is true
`-q, --quiet`	Quiet startup: no banner, suppress REPL warnings
`--banner={yes|no|auto}`	Enable or disable startup banner
`--color={yes|no|auto}`	Enable or disable color text
`--history-file={yes|no}`	Load or save history
`--depwarn={yes|no|error}`	Enable or disable syntax and method deprecation warnings (`error` turns warnings into errors)
`--warn-overwrite={yes|no}`	Enable or disable method overwrite warnings
`-C, --cpu-target <target>`	Limit usage of cpu features up to `<target>`; set to `help` to see the available options
`-O, --optimize={0,1,2,3}`	Set the optimization level (default level is 2 if unspecified or 3 if used without a level)
`-g, -g <level>`	Enable / Set the level of debug info generation (default level is 1 if unspecified or 2 if used without a level)
`--inline={yes|no}`	Control whether inlining is permitted, including overriding `@inline` declarations
`--check-bounds={yes|no}`	Emit bounds checks always or never (ignoring declarations)
`--math-mode={ieee,fast}`	Disallow or enable unsafe floating point optimizations (overrides @fastmath declaration)
`--code-coverage={none|user|all}`	Count executions of source lines
`--code-coverage`	equivalent to `--code-coverage=user`
`--track-allocation={none|user|all}`	Count bytes allocated by each source line
`--track-allocation`	equivalent to `--track-allocation=user`

Chapter 2

Variables

A variable, in Julia, is a name associated (or bound) to a value. It's useful when you want to store a value (that you obtained after some math, for example) for later use. For example:

```
# Assign the value 10 to the variable x
julia> x = 10
10

# Doing math with x's value
julia> x + 1
11

# Reassign x's value
julia> x = 1 + 1
2

# You can assign values of other types, like strings of text
julia> x = "Hello World!"
"Hello World!"
```

Julia provides an extremely flexible system for naming variables. Variable names are case-sensitive, and have no semantic meaning (that is, the language will not treat variables differently based on their names).

```
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 = ""
""
```

Unicode names (in UTF-8 encoding) are allowed:

```
julia> δ = 0.00001
1.0e-5

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

In the Julia REPL and several other Julia editing environments, you can type many Unicode math symbols by typing the backslashed LaTeX symbol name followed by tab. For example, the variable name δ can be entered by typing `\delta`-*tab*, or even α by `\alpha`-*tab*-`\hat`-*tab*-`\_2`-*tab*. (If you find a symbol somewhere, e.g. in someone else's code, that you don't know how to type, the REPL help will tell you: just type ? and then paste the symbol.)

Julia will even let you redefine built-in constants and functions if needed (although this is not recommended to avoid potential confusions):

```
julia> pi = 3
3

julia> pi
3

julia> sqrt = 4
4
```

However, if you try to redefine a built-in constant or function already in use, Julia will give you an error:

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
ERROR: cannot assign variable MathConstants.pi from module Main

julia> sqrt(100)
10.0

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

2.1   Allowed Variable Names

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like + are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example (+) refers to the addition function, and (+) = f will reassign it. Most of the Unicode infix operators (in category Sm), such as , are parsed as infix operators and are available for user-defined methods (e.g. you can use `const  = kron` to define as an infix Kronecker product). Operators can also be suffixed with modifying marks, primes, and sub/superscripts, e.g. +" is parsed as an infix operator with the same precedence as +.

The only explicitly disallowed names for variables are the names of built-in statements:

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

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

Some Unicode characters are considered to be equivalent in identifiers. Different ways of entering Unicode combining characters (e.g., accents) are treated as equivalent (specifically, Julia identifiers are NFC-normalized). The Unicode characters (U+025B: Latin small letter open e) and µ (U+00B5: micro sign) are treated as equivalent to the corresponding Greek letters, because the former are easily accessible via some input methods.

2.2 Stylistic Conventions

While Julia imposes few restrictions on valid names, it has become useful to adopt the following conventions:

- Names of variables are in lower case.

- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.

- Names of `Types` and `Modules` begin with a capital letter and word separation is shown with upper camel case instead of underscores.

- Names of `functions` and `macros` are in lower case, without underscores.

- Functions that write to their arguments have names that end in `!`. These are sometimes called "mutating" or "in-place" functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

For more information about stylistic conventions, see the Style Guide.

Chapter 3

Integers and Floating-Point Numbers

Integers and floating-point values are the basic building blocks of arithmetic and computation. Built-in representations of such values are called numeric primitives, while representations of integers and floating-point numbers as immediate values in code are known as numeric literals. For example, 1 is an integer literal, while 1.0 is a floating-point literal; their binary in-memory representations as objects are numeric primitives.

Julia provides a broad range of primitive numeric types, and a full complement of arithmetic and bitwise operators as well as standard mathematical functions are defined over them. These map directly onto numeric types and operations that are natively supported on modern computers, thus allowing Julia to take full advantage of computational resources. Additionally, Julia provides software support for Arbitrary Precision Arithmetic, which can handle operations on numeric values that cannot be represented effectively in native hardware representations, but at the cost of relatively slower performance.

The following are Julia's primitive numeric types:

- **Integer types:**

Type	Signed?	Number of bits	Smallest value	Largest value
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)

- **Floating-point types:**

Additionally, full support for Complex and Rational Numbers is built on top of these primitive numeric types. All numeric types interoperate naturally without explicit casting, thanks to a flexible, user-extensible type promotion system.

Type	Precision	Number of bits
Float16	half	16
Float32	single	32
Float64	double	64

3.1   Integers

Literal integers are represented in the standard manner:

```julia
julia> 1
1

julia> 1234
1234
```

The default type for an integer literal depends on whether the target system has a 32-bit architecture or a 64-bit architecture:

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

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

The Julia internal variable Sys.WORD_SIZE indicates whether the target system is 32-bit or 64-bit:

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

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

Julia also defines the types Int and UInt, which are aliases for the system's signed and unsigned native integer types respectively:

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

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

Larger integer literals that cannot be represented using only 32 bits but can be represented in 64 bits always create 64-bit integers, regardless of the system type:

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

Unsigned integers are input and output using the `0x` prefix and hexadecimal (base 16) digits `0-9a-f` (the capitalized digits `A-F` also work for input). The size of the unsigned value is determined by the number of hex digits used:

```
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> 0x11112222333344445555666677778888
0x11112222333344445555666677778888

julia> typeof(ans)
UInt128
```

This behavior is based on the observation that when one uses unsigned hex literals for integer values, one typically is using them to represent a fixed numeric byte sequence, rather than just an integer value.

Recall that the variable `ans` is set to the value of the last expression evaluated in an interactive session. This does not occur when Julia code is run in other ways.

Binary and octal literals are also supported:

```
julia> 0b10
0x02

julia> typeof(ans)
UInt8
```

```
julia> 0o010
0x08

julia> typeof(ans)
UInt8

julia> 0x00000000000000001111222233334444
0x00000000000000001111222233334444

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
```

The minimum and maximum representable values of primitive numeric types such as integers are given by the typemin and typemax functions:

```
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]
```

The values returned by typemin and typemax are always of the given argument type. (The above expression uses several features we have yet to introduce, including for loops, Strings, and Interpolation, but should be easy enough to understand for users with some existing programming experience.)

Overflow behavior

In Julia, exceeding the maximum representable value of a given type results in a wraparound behavior:

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

julia> x + 1
-9223372036854775808

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

Thus, arithmetic with Julia integers is actually a form of modular arithmetic. This reflects the characteristics of the underlying arithmetic of integers as implemented on modern computers. In applications where overflow is possible, explicit checking for wraparound produced by overflow is essential; otherwise, the BigInt type in Arbitrary Precision Arithmetic is recommended instead.

Division errors

Integer division (the div function) has two exceptional cases: dividing by zero, and dividing the lowest negative number (typemin) by -1. Both of these cases throw a DivideError. The remainder and modulus functions (rem and mod) throw a DivideError when their second argument is zero.

3.2 Floating-Point Numbers

Literal floating-point numbers are represented in the standard formats, using E-notation when necessary:

```
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
```

The above results are all Float64 values. Literal Float32 values can be entered by writing an f in place of e:

```
julia> 0.5f0
0.5f0

julia> typeof(ans)
Float32
```

```
julia> 2.5f-4
0.00025f0
```

Values can be converted to `Float32` easily:

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

julia> typeof(ans)
Float32
```

Hexadecimal floating-point literals are also valid, but only as `Float64` values, with p preceding the base-2 exponent:

```
julia> 0x1p0
1.0

julia> 0x1.8p3
12.0

julia> 0x.4p-1
0.125

julia> typeof(ans)
Float64
```

Half-precision floating-point numbers are also supported (`Float16`), but they are implemented in software and use `Float32` for calculations.

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

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

The underscore _ can be used as digit separator:

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

Floating-point zero

Floating-point numbers have two zeros, positive zero and negative zero. They are equal to each other but have different binary representations, as can be seen using the `bitstring` function:

```
julia> 0.0 == -0.0
true

julia> bitstring(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

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

Special floating-point values

There are three specified standard floating-point values that do not correspond to any point on the real number line:

Float16	Float32	Float64	Name	Description
Inf16	Inf32	Inf	positive infinity	a value greater than all finite floating-point values
-Inf16	-Inf32	-Inf	negative infinity	a value less than all finite floating-point values
NaN16	NaN32	NaN	not a number	a value not == to any floating-point value (including itself)

For further discussion of how these non-finite floating-point values are ordered with respect to each other and other floats, see Numeric Comparisons. By the IEEE 754 standard, these floating-point values are the results of certain arithmetic operations:

```julia
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
```

The typemin and typemax functions also apply to floating-point types:

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

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

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

Machine epsilon

Most real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers, which is often known as machine epsilon.

Julia provides `eps`, which gives the distance between `1.0` and the next larger representable floating-point value:

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

julia> eps(Float64)
2.220446049250313e-16

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

These values are `2.0^-23` and `2.0^-52` as `Float32` and `Float64` values, respectively. The `eps` function can also take a floating-point value as an argument, and gives the absolute difference between that value and the next representable floating point value. That is, `eps(x)` yields a value of the same type as x such that `x + eps(x)` is the next representable floating-point value larger than x:

```
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
```

The distance between two adjacent representable floating-point numbers is not constant, but is smaller for smaller values and larger for larger values. In other words, the representable floating-point numbers are densest in the real number line near zero, and grow sparser exponentially as one moves farther away from zero. By definition, `eps(1.0)` is the same as `eps(Float64)` since `1.0` is a 64-bit floating-point value.

Julia also provides the `nextfloat` and `prevfloat` functions which return the next largest or smallest representable floating-point number to the argument respectively:

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0
```

```julia
julia> prevfloat(x)
1.2499999f0

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

julia> bitstring(x)
"00111111101000000000000000000000"

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

This example highlights the general principle that the adjacent representable floating-point numbers also have adjacent binary integer representations.

Rounding modes

If a number doesn't have an exact floating-point representation, it must be rounded to an appropriate representable value. However, the manner in which this rounding is done can be changed if required according to the rounding modes presented in the IEEE 754 standard.

The default mode used is always `RoundNearest`, which rounds to the nearest representable value, with ties rounded towards the nearest value with an even least significant bit.

Background and References

Floating-point arithmetic entails many subtleties which can be surprising to users who are unfamiliar with the low-level implementation details. However, these subtleties are described in detail in most books on scientific computation, and also in the following references:

- The definitive guide to floating point arithmetic is the IEEE 754-2008 Standard; however, it is not available for free online.

- For a brief but lucid presentation of how floating-point numbers are represented, see John D. Cook's article on the subject as well as his introduction to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers.

- Also recommended is Bruce Dawson's series of blog posts on floating-point numbers.

- For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper What Every Computer Scientist Should Know About Floating-Point Arithmetic.

- For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the collected writings of William Kahan, commonly known as the "Father of Floating-Point". Of particular interest may be An Interview with the Old Man of Floating-Point.

3.3   Arbitrary Precision Arithmetic

To allow computations with arbitrary-precision integers and floating point numbers, Julia wraps the GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library, respectively. The `BigInt` and `BigFloat` types are available in Julia for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, and parse can be used to construct them from AbstractStrings. Once created, they participate in arithmetic with all other numeric types thanks to Julia's type promotion and conversion mechanism:

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

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

julia> parse(BigFloat, "1.23456789012345678901")
1.234567890123456789010000000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.459565876494606882133333333333333333333333333333333333333333333333333333333344e+19

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

However, type promotion between the primitive types above and BigInt/BigFloat is not automatic and must be explicitly stated.

```julia
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
```

The default precision (in number of bits of the significand) and rounding mode of BigFloat operations can be changed globally by calling setprecision and setrounding, and all further calculations will take these changes in account. Alternatively, the precision or the rounding can be changed only within the execution of a particular block of code by using the same functions with a do block:

```julia
julia> setrounding(BigFloat, RoundUp) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.100000000000000000000000000000000000000000000000000000000000000000000000000003

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

```
1.099999999999999999999999999999999999999999999999999999999999999999999999999986

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

3.4 Numeric Literal Coefficients

To make common numeric formulae and expressions clearer, Julia allows variables to be immediately preceded by a numeric literal, implying multiplication. This makes writing polynomial expressions much cleaner:

```
julia> x = 3
3

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

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

It also makes writing exponential functions more elegant:

```
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).

Numeric literals also work as coefficients to parenthesized expressions:

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

> **Note**
>
> The precedence of numeric literal coefficients used for implicit multiplication is higher than other binary operators such as multiplication (*), and division (/, \, and //). This means, for example, that 1 / 2im equals -0.5im and 6 // 2(2 + 1) equals 1 // 1.

Additionally, parenthesized expressions can be used as coefficients to variables, implying multiplication of the expression by the variable:

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

Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication:

```
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
```

Both expressions are interpreted as function application: any expression that is not a numeric literal, when immediately followed by a parenthetical, is interpreted as a function applied to the values in parentheses (see Functions for more about functions). Thus, in both of these cases, an error occurs since the left-hand value is not a function.

The above syntactic enhancements significantly reduce the visual noise incurred when writing common mathematical formulae. Note that no whitespace may come between a numeric literal coefficient and the identifier or parenthesized expression which it multiplies.

Syntax Conflicts

Juxtaposed literal coefficient syntax may conflict with two numeric literal syntaxes: hexadecimal integer literals and engineering notation for floating-point literals. Here are some situations where syntactic conflicts arise:

- The hexadecimal integer literal expression `0xff` could be interpreted as the numeric literal `0` multiplied by the variable `xff`.

- The floating-point literal expression `1e10` could be interpreted as the numeric literal `1` multiplied by the variable `e10`, and similarly with the equivalent E form.

- The 32-bit floating-point literal expression `1.5f22` could be interpreted as the numeric literal `1.5` multiplied by the variable `f22`.

In all cases, we resolve the ambiguity in favor of interpretation as numeric literals:

- Expressions starting with `0x` are always hexadecimal literals.

- Expressions starting with a numeric literal followed by `e` or `E` are always floating-point literals.

- 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`.

3.5   Literal zero and one

Julia provides functions which return literal 0 and 1 corresponding to a specified type or the type of a given variable.

Function	Description
zero(x)	Literal zero of type x or type of variable x
one(x)	Literal one of type x or type of variable x

These functions are useful in Numeric Comparisons to avoid overhead from unnecessary type conversion.

Examples:

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
0.0

julia> one(Int32)
1

julia> one(BigFloat)
1.0
```

Chapter 4

Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

4.1 Arithmetic Operators

The following arithmetic operators are supported on all primitive numeric types:

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x ÷ y`	integer divide	x / y, truncated to an integer
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises x to the yth power
`x % y`	remainder	equivalent to `rem(x,y)`

as well as the negation on Bool types:

Expression	Name	Description
`!x`	negation	changes `true` to `false` and vice versa

Julia's promotion system makes arithmetic operations on mixtures of argument types "just work" naturally and automatically. See Conversion and Promotion for details of the promotion system.

Here are some simple examples using arithmetic operators:

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(By convention, we tend to space operators more tightly if they get applied before other nearby operators. For instance, we would generally write -x + 2 to reflect that first x gets negated, and then 2 is added to that result.)

4.2   Bitwise Operators

The following bitwise operators are supported on all primitive integer types:

Expression	Name
~x	bitwise not
x & y	bitwise and
x | y	bitwise or
x y	bitwise xor (exclusive or)
x >>> y	logical shift right
x >> y	arithmetic shift right
x << y	logical/arithmetic shift left

Here are some examples with bitwise operators:

```julia
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
```

4.3   Updating operators

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. The updating version of the binary operator is formed by placing a = immediately after the operator. For example, writing x += 3 is equivalent to writing x = x + 3:

```julia
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

The updating versions of all the binary arithmetic and bitwise operators are:

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

> **Note**
>
> An updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.
>
> ```julia
> julia> x = 0x01; typeof(x)
> UInt8
>
> julia> x *= 2 # Same as x = x * 2
> 2
>
> julia> typeof(x)
> Int64
> ```

4.4 Vectorized "dot" operators

For *every* binary operation like `^`, there is a corresponding "dot" operation `.^` that is *automatically* defined to perform `^` element-by-element on arrays. For example, `[1,2,3] ^ 3` is not defined, since there is no standard mathematical meaning to "cubing" a (non-square) array, but `[1,2,3] .^ 3` is defined as computing the elementwise (or "vectorized") result `[1^3, 2^3, 3^3]`. Similarly for unary operators like `!` or `√`, there is a corresponding `.√` that applies the operator elementwise.

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

More specifically, a `.^ b` is parsed as the "dot" call `(^).(a,b)`, which performs a broadcast operation: it can combine arrays and scalars, arrays of the same size (performing the operation elementwise), and even arrays of different shapes (e.g. combining row and column vectors to produce a matrix). Moreover, like all vectorized "dot calls," these "dot operators" are *fusing*. For example, if you compute `2 .* A.^2 .+ sin.(A)` (or equivalently `@. 2A^2 + sin(A)`, using the `@.` macro) for an array A, it performs a *single* loop over A, computing `2a^2 + sin(a)` for each element of A. In particular, nested dot calls like `f.(g.(x))` are fused, and "adjacent" binary operators like `x .+ 3 .* x.^2` are equivalent to nested dot calls `(+).(x, (*).(3, (^).(x, 2)))`.

Furthermore, "dotted" updating operators like `a .+= b` (or `@. a += b`) are parsed as `a .= a .+ b`, where `.=` is a fused *in-place* assignment operation (see the dot syntax documentation).

Note the dot syntax is also applicable to user-defined operators. For example, if you define `⊗(A,B) = kron(A,B)` to give a convenient infix syntax `A ⊗ B` for Kronecker products (kron), then `[A,B] .⊗ [C,D]` will compute `[A⊗C, B⊗D]` with no additional coding.

Combining dot operators with numeric literals can be ambiguous. For example, it is not clear whether `1.+x` means `1. + x` or `1 .+ x`. Therefore this syntax is disallowed, and spaces must be used around the operator in such cases.

4.5 Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

Here are some simple examples:

Operator	Name
==	equality
!=, ≠	inequality
<	less than
<=, ≤	less than or equal to
>	greater than
>=, ≥	greater than or equal to

```
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
```

Integers are compared in the standard manner – by comparison of bits. Floating-point numbers are compared according to the IEEE 754 standard:

- Finite numbers are ordered in the usual manner.

- Positive zero is equal but not greater than negative zero.

- `Inf` is equal to itself and greater than everything else except `NaN`.

- `-Inf` is equal to itself and less then everything else except `NaN`.

- `NaN` is not equal to, not less than, and not greater than anything, including itself.

The last point is potentially surprising and thus worth noting:

```julia
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

and can cause especial headaches with arrays:

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

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons:

Function	Tests if
isequal(x, y)	x and y are identical
isfinite(x)	x is a finite number
isinf(x)	x is infinite
isnan(x)	x is not a number

isequal considers NaNs equal to each other:

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

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

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

isequal can also be used to distinguish signed zeros:

```julia
julia> -0.0 == 0.0
true

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

Mixed-type comparisons between signed integers, unsigned integers, and floats can be tricky. A great deal of care has been taken to ensure that Julia does them correctly.

For other types, isequal defaults to calling ==, so if you want to define equality for your own types then you only need to add a == method. If you define your own equality function, you should probably define a corresponding hash method to ensure that isequal(x,y) implies hash(x) == hash(y).

Chaining comparisons

Unlike most languages, with the notable exception of Python, comparisons can be arbitrarily chained:

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

Chaining comparisons is often quite convenient in numerical code. Chained comparisons use the && operator for scalar comparisons, and the & operator for elementwise comparisons, which allows them to work on arrays. For example, 0 .< A .< 1 gives a boolean array whose entries are true where the corresponding elements of A are between 0 and 1.

Note the evaluation behavior of chained comparisons:

```julia
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
```

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as v(1) < v(2) && v(2) <= v(3). However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit && operator should be used explicitly (see Short-Circuit Evaluation).

Elementary Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complex numbers, wherever such definitions make sense.

Moreover, these functions (like any Julia function) can be applied in "vectorized" fashion to arrays and other collections with the dot syntax f.(A), e.g. sin.(A) will compute the sine of each element of an array A.

4.6   Operator Precedence and Associativity

Julia applies the following order and associativity of operations, from highest precedence to lowest:

For a complete list of *every* Julia operator's precedence, see the top of this file: src/julia-parser.scm

You can also find the numerical precedence for any given operator via the built-in function Base.operator_precedence, where higher numbers take precedence:

---

[1]The unary operators + and - require explicit parentheses around their argument to disambiguate them from the operator ++, etc. Other compositions of unary operators are parsed with right-associativity, e. g., √√-a as √(√(-a)).

[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.

Category	Operators	Associativity
Syntax	`.` followed by `::`	Left
Exponentiation	`^`	Right
Unary	`+ - √`	Right[1]
Bitshifts	`<< >> >>>`	Left
Fractions	`//`	Left
Multiplication	`* / % & \ ÷`	Left[2]
Addition	`+ - |`	Left[2]
Syntax	`: ..`	Left
Syntax	`|>`	Left
Syntax	`<|`	Right
Comparisons	`> < >= <= == === != !== <:`	Non-associative
Control flow	`&&` followed by `||` followed by `?`	Right
Pair	`=>`	Right
Assignments	`= += -= *= /= //= \= ^= ÷= %= |= &= = <<= >>= >>>=`	Right

```julia
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)
```

A symbol representing the operator associativity can also be found by calling the built-in function `Base.operator_associativity`:

```julia
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)
```

Note that symbols such as `:sin` return precedence 0. This value represents invalid operators and not operators of lowest precedence. Similarly, such operators are assigned associativity `:none`.

4.7 Numerical Conversions

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

- The notation `T(x)` or `convert(T,x)` converts x to a value of type T.

  - If T is a floating-point type, the result is the nearest representable value, which could be positive or negative infinity.

  - If T is an integer type, an `InexactError` is raised if x is not representable by T.

- `x % T` converts an integer x to a value of integer type T congruent to x modulo 2^n, where n is the number of bits in T. 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
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(Int8, 3.14)
Stacktrace:
[...]

julia> Int8(128.0)
ERROR: InexactError: Int8(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 functions

Function	Description	Return type
`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

Function	Description
`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,...

Division functions

Sign and absolute value functions

Function	Description
`abs(x)`	a positive value with the magnitude of x
`abs2(x)`	the squared magnitude of x
`sign(x)`	indicates the sign of x, returning -1, 0, or +1
`signbit(x)`	indicates whether the sign bit is on (true) or off (false)
`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

Powers, logs and roots

Function	Description
`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

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.

Trigonometric and hyperbolic functions

All the standard trigonometric and hyperbolic functions are also defined:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

These are all single-argument functions, with `atan` also accepting two arguments corresponding to a traditional `atan2` function.

Additionally, `sinpi(x)` and `cospi(x)` are provided for more accurate computations of `sin(pi*x)` and `cos(pi*x)` respectively.

In order to compute trigonometric functions with degrees instead of radians, suffix the function with d. For example, `sind(x)` computes the sine of x where x is specified in degrees. The complete list of trigonometric functions with degree variants is:

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

Special functions

Many other special mathematical functions are provided by the package SpecialFunctions.jl.

Chapter 5

Complex and Rational Numbers

Julia ships with predefined types representing both complex and rational numbers, and supports all standard Mathematical Operations and Elementary Functions on them. Conversion and Promotion are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

5.1 Complex Numbers

The global constant `im` is bound to the complex number *i*, representing the principal square root of -1. It was deemed harmful to co-opt the name `i` for a global constant, since it is such a popular index variable name. Since Julia allows numeric literals to be juxtaposed with identifiers as coefficients, this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

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

You can perform all the standard arithmetic operations with complex numbers:

```julia
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
```

The promotion mechanism ensures that combinations of operands of different types just work:

```
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
```

Note that 3/4im == 3/(4*im) == -(3/4*im), since a literal coefficient binds more tightly than division.

Standard functions to manipulate complex values are provided:

```
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
```

As usual, the absolute value (abs) of a complex number is its distance from zero. abs2 gives the square of the absolute value, and is of particular use for complex numbers where it avoids taking a square root. angle returns the phase angle in radians (also known as the *argument* or *arg* function). The full gamut of other Elementary Functions is also defined for complex numbers:

```
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
```

Note that mathematical functions typically return real values when applied to real numbers and complex values when applied to complex numbers. For example, sqrt behaves differently when applied to -1 versus -1 + 0im even though -1 == -1 + 0im:

```
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
```

The literal numeric coefficient notation does not work when constructing a complex number from variables. Instead, the multiplication must be explicitly written out:

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

However, this is *not* recommended; Use the complex function instead to construct a complex value directly from its real and imaginary parts:

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

This construction avoids the multiplication and addition operations.

Inf and NaN propagate through complex numbers in the real and imaginary parts of a complex number as described in the Special floating-point values section:

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

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

5.2   Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the // operator:

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

If the numerator and denominator of a rational have common factors, they are reduced to lowest terms such that the denominator is non-negative:

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

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

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

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

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational value can be extracted using the numerator and denominator functions:

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

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

Direct comparison of the numerator and denominator is generally not necessary, since the standard arithmetic and comparison operations are defined for rational values:

```
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
```

Rationals can be easily converted to floating-point numbers:

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

Conversion from rational to floating-point respects the following identity for any integral values of a and b, with the exception of the case a == 0 and b == 0:

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

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

Constructing infinite rational values is acceptable:

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

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

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

Trying to construct a NaN rational value, however, is not:

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

As usual, the promotion system makes interactions with other numeric types effortless:

```
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 6

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.

6.1   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
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
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
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
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
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
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
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)
```

6.2   String Basics

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

```julia
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
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 is returned by `firstindex(str)`, and the index of the last element with `lastindex(str)`. The keyword end can be used inside an indexing operation as shorthand for the last index along the given dimension. Most indexing in Julia is 1-based: the first element of many integer-indexed objects is found at index 1. (As we will see below, this does not necessarily mean that the last element is found at index n, where n is the length of the string.)

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 "Hello, world.\n"
  at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access "Hello, world.\n"
  at index [15]
Stacktrace:
[...]
```

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
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.

6.3 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
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. 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. This means that not every byte index into a UTF-8 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
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.

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

```julia
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
```

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 implies that overlong and too high code unit sequences are accepted. This rule is best explained by an example:

```julia
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}:
 false
 false
 false
  true

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.

6.4 Concatenation

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

```julia
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
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
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.

6.5 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
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 a concatenation of string literals with variables.

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

Both concatenation and string interpolation call `string` to convert objects into string form. Most non-`AbstractString` objects are converted to strings closely corresponding to how they are entered as literal expressions:

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

6.6 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
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
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.

```julia
"""hello"""
```

is equivalent to

```julia
"""
hello"""
```

but

```julia
"""

hello"""
```

will contain a literal newline at the beginning.

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

```julia
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".

6.7   Common Operations

You can lexicographically compare strings using the standard comparison operators:

```julia
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` function:

```julia
julia> findfirst(isequal('x'), "xylophone")
1

julia> findfirst(isequal('p'), "xylophone")
5

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

You can start the search for a character at a given offset by using `findnext` with a third argument:

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

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

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

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

```julia
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
julia> repeat(".:Z:.", 10)
".:Z:..: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 nth character starting after index `i`.

- `prevind(str, i, n=1)` find the start of the nth character starting before index `i`.

6.8   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.

6.9   Regular Expressions

Julia has Perl-compatible regular expressions (regexes), as provided by the PCRE library. 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
julia> r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

julia> typeof(ans)
Regex
```

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

```julia
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
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:

```julia
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
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
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
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
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
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).

6.10   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
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
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
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.

6.11   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.

6.12   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 7

Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, in the sense that functions can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

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

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "assignment form":

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

In the assignment form, the body of the function must be a single expression, although it can be a compound expression (see Compound Expressions). Short, simple function definitions are common in Julia. The short function syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

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

Without parentheses, the expression f refers to the function object, and can be passed around like any value:

```julia
julia> g = f;

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

As with variables, Unicode can also be used for function names:

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

julia> ∑(2, 3)
5
```

7.1   Argument Passing Behavior

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable *bindings* (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as `Arrays`) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

7.2   The `return` Keyword

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, `f`, from the previous section this is the value of the expression `x + y`. As in C and most other imperative or functional languages, the `return` keyword causes a function to return immediately, providing an expression whose value is returned:

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

Since function definitions can be entered into interactive sessions, it is easy to compare these definitions:

```julia
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
```

Of course, in a purely linear function body like g, the usage of `return` is pointless since the expression `x + y` is never evaluated and we could simply make `x * y` the last expression in the function and omit the `return`. In conjunction with other control flow, however, `return` is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length x and y, avoiding overflow:

```julia
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
```

There are three possible points of return from this function, returning the values of three different expressions, depending on the values of x and y. The `return` on the last line could be omitted since it is the last expression.

A return type can also be specified in the function declaration using the `::` operator. This converts the return value to the specified type.

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

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

This function will always return an `Int8` regardless of the types of x and y. See Type Declarations for more on return types.

7.3　Operators Are Functions

In Julia, most operators are just functions with support for special syntax. (The exceptions are operators with special evaluation semantics like && and ||. These operators cannot be functions since Short-Circuit Evaluation requires that their operands are not evaluated before evaluation of the operator.) Accordingly, you can also apply them using parenthesized argument lists, just as you would any other function:

```
julia> 1 + 2 + 3
6

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

The infix form is exactly equivalent to the function application form – in fact the former is parsed to produce the function call internally. This also means that you can assign and pass around operators such as + and * just like you would with other function values:

```
julia> f = +;

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

Under the name f, the function does not support infix notation, however.

7.4　Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names. These are:

Expression	Calls
`[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!`

7.5   Anonymous Functions

Functions in Julia are first-class objects: they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

```
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)
```

This creates a function taking one argument x and returning the value of the polynomial `x^2 + 2x - 1` at that value. Notice that the result is a generic function, but with a compiler-generated name based on consecutive numbering.

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is map, which applies a function to each value of an array and returns a new array containing the resulting values:

```
julia> map(round, [1.2,3.5,1.7])
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

This is fine if a named function effecting the transform already exists to pass as the first argument to map. Often, however, a ready-to-use, named function does not exist. In these situations, the anonymous function construct allows easy creation of a single-use function object without needing a name:

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

An anonymous function accepting multiple arguments can be written using the syntax `(x,y,z)->2x+y-z`. A zero-argument anonymous function is written as `()->3`. The idea of a function with no arguments may seem strange, but is useful for "delaying" a computation. In this usage, a block of code is wrapped in a zero-argument function, which is later invoked by calling it as `f`.

7.6   Tuples

Julia has a built-in data structure called a *tuple* that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is *immutable*). Tuples are constructed with commas and parentheses, and can be accessed via indexing:

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

julia> (1,)
(1,)

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

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

Notice that a length-1 tuple must be written with a comma, (1,), since (1) would just be a parenthesized value. ( ) represents the empty (length-0) tuple.

7.7   Named Tuples

The components of tuples can optionally be named, in which case a *named tuple* is constructed:

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

julia> x.a
1
```

Named tuples are very similar to tuples, except that fields can additionally be accessed by name using dot syntax (x . a).

7.8   Multiple Return Values

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

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

If you call it in an interactive session without assigning the return value anywhere, you will see the tuple returned:

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

A typical usage of such a pair of return values, however, extracts each value into a variable. Julia supports simple tuple "destructuring" that facilitates this:

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

julia> x
5

julia> y
6
```

You can also return multiple values via an explicit usage of the `return` keyword:

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

This has the exact same effect as the previous definition of `foo`.

7.9   Argument destructuring

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
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.

7.10   Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". You can define a varargs function by following the last argument with an ellipsis:

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

The variables a and b are bound to the first two argument values as usual, and the variable x is bound to an iterable collection of the zero or more values passed to `bar` after its first two arguments:

```julia
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))
```

In all these cases, x is bound to a tuple of the trailing values passed to bar.

It is possible to constrain the number of values passed as a variable argument; this will be discussed later in Parametrically-constrained Varargs methods.

On the flip side, it is often handy to "splat" the values contained in an iterable collection into a function call as individual arguments. To do this, one also uses ... but in the function call instead:

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

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

In this case a tuple of values is spliced into a varargs call precisely where the variable number of arguments go. This need not be the case, however:

```
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))
```

Furthermore, the iterable object splatted into a function call need not be a tuple:

```
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))
```

Also, the function that arguments are splatted into need not be a varargs function (although it often is):

```julia
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
```

As you can see, if the wrong number of elements are in the splatted container, then the function call will fail, just as it would if too many arguments were given explicitly.

7.11   Optional Arguments

In many cases, function arguments have sensible default values and therefore might not need to be passed explicitly in every call. For example, the function `Date(y, [m, d])` from `Dates` module constructs a `Date` type for a given year y, month m and day d. However, m and d arguments are optional and their default value is 1. This behavior can be expressed concisely as:

```julia
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
```

Observe, that this definition calls another method of `Date` function that takes one argument of `UTInstant{Day}` type.

With this definition, the function can be called with either one, two or three arguments, and 1 is automatically passed when any of the arguments is not specified:

```julia
julia> using Dates

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

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

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

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see Note on Optional and keyword Arguments). This can be checked for our `Date` function example by calling `methods` function.

7.12 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
```

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)
```

Inside `f`, `kwargs` will be a named tuple. Named tuples (as well as dictionaries) can be passed as keyword arguments using a semicolon in a call, e.g. `f(x, z=1; kwargs...)`.

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.

7.13   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.

7.14   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 Control Flow.)

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.

7.15 Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have "vectorized" versions of functions, which simply apply a given function f(x) to each element of an array A to yield a new array via f(A). This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: if loops are slow, the "vectorized" version of a function can call fast library code written in a low-level language. In Julia, vectorized functions are *not* required for performance, and indeed it is often beneficial to write your own loops (see Performance Tips), but they can still be convenient. Therefore, *any* Julia function f can be applied elementwise to any array (or other collection) with the syntax f.(A). For example sin can be applied to all elements in the vector A, like so:

```
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
```

Of course, you can omit the dot if you write a specialized "vector" method of f, e.g. via `f(A::AbstractArray) = map(f, A)`, and this is just as efficient as `f.(A)`. But that approach requires you to decide in advance which functions you want to vectorize.

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
```

7.16   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 8

Control Flow

Julia provides a variety of control flow constructs:

- Compound Expressions: `begin` and `(;)`.

- Conditional Evaluation: `if-elseif-else` and `?:` (ternary operator).

- Short-Circuit Evaluation: `&&`, `||` and chained comparisons.

- Repeated Evaluation: Loops: `while` and `for`.

- Exception Handling: `try-catch`, `error` and `throw`.

- Tasks (aka Coroutines): `yieldto`.

The first five control flow mechanisms are standard to high-level programming languages. `Task`s are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

8.1 Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: `begin` blocks and `(;)` chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `begin` block:

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

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the `(;)` chain syntax comes in handy:

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

This syntax is particularly useful with the terse single-line function definition form introduced in Functions. Although it is typical, there is no requirement that begin blocks be multiline or that (;) chains be single-line:

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

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

8.2   Conditional Evaluation

Conditional evaluation 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:

```julia
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. Here it is in action:

```julia
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
```

The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used. The condition expressions in the `if-elseif-else` construct are evaluated until the first one evaluates to `true`, after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

`if` blocks are "leaky", i.e. they do not introduce a local scope. This means that new variables defined inside the `if` clauses can be used after the `if` block, even if they weren't defined before. So, we could have defined the `test` function above as

```julia
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.
```

The variable `relation` is declared inside the `if` block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error

```julia
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` blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so

```julia
julia> x = 3
3

julia> if x > 0
           "positive!"
       else
```

```
            "negative..."
        end
"positive!"
```

Note that very short conditional statements (one-liners) are frequently expressed using Short-Circuit Evaluation in Julia, as outlined in the next section.

Unlike C, MATLAB, Perl, Python, and Ruby – but like Java, and a few other stricter, typed languages – it is an error if the value of a conditional expression is anything but `true` or `false`:

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

This error indicates that the conditional was of the wrong type: `Int64` rather than the required `Bool`.

The so-called "ternary operator", `? :`, is closely related to the `if-elseif-else` syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

```
a ? b : c
```

The expression a, before the `?`, is a condition expression, and the ternary operation evaluates the expression b, before the `:`, if the condition a is `true` or the expression c, after the `:`, if it is `false`. Note that the spaces around `?` and `:` are mandatory: an expression like a?b:c is not a valid ternary expression (but a newline is acceptable after both the `?` and the `:`).

The easiest way to understand this behavior is to see an example. In the previous example, the `println` call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```julia
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
```

If the expression `x < y` is true, the entire ternary operator expression evaluates to the string `"less than"` and otherwise it evaluates to the string `"not less than"`. The original three-way example requires chaining multiple uses of the ternary operator together:

```julia
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
```

To facilitate chaining, the operator associates from right to left.

It is significant that like `if-elseif-else`, the expressions before and after the `:` are only evaluated if the condition expression evaluates to `true` or `false`, respectively:

```
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"
```

8.3 Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the && and || boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

- In the expression a && b, the subexpression b is only evaluated if a evaluates to `true`.

- In the expression a || b, the subexpression b is only evaluated if a evaluates to `false`.

The reasoning is that a && b must be `false` if a is `false`, regardless of the value of b, and likewise, the value of a || b must be true if a is `true`, regardless of the value of b. Both && and || associate to the right, but && has higher precedence than || does. It's easy to experiment with this behavior:

```
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
```

You can easily experiment in the same way with the associativity and precedence of various combinations of && and || operators.

This behavior is frequently used in Julia to form an alternative to very short if statements. Instead of if <cond> <statement> end, one can write <cond> && <statement> (which could be read as: <cond> *and then* <statement>). Similarly, instead of if ! <cond> <statement> end, one can write <cond> || <statement> (which could be read as: <cond> *or else* <statement>).

For example, a recursive factorial routine could be defined like this:

```
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
```

Boolean operations *without* short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: & and |. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

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

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

Just like condition expressions used in if, elseif or the ternary operator, the operands of && or || must be boolean values (true or false). Using a non-boolean value anywhere except for the last entry in a conditional chain is an error:

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

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

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

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

8.4 Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the while loop and the for loop. Here is an example of a while loop:

```julia
julia> i = 1;

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

The while loop evaluates the condition expression (i <= 5 in this case), and as long it remains true, keeps also evaluating the body of the while loop. If the condition expression is false when the while loop is first reached, the body is never evaluated.

The for loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above while loop does is so common, it can be expressed more concisely with a for loop:

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

Here the `1:5` is a range object, representing the sequence of numbers 1, 2, 3, 4, 5. The `for` loop iterates through these values, assigning each one in turn to the variable `i`. One rather important distinction between the previous `while` loop form and the `for` loop form is the scope during which the variable is visible. If the variable `i` has not been introduced in another scope, in the `for` loop form, it is visible only inside of the `for` loop, and not outside/afterwards. You'll either need a new interactive session instance or a different variable name to test this:

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

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

See Scope of Variables for a detailed explanation of variable scope and how it works in Julia.

In general, the `for` loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword `in` or is typically used instead of =, since it makes the code read more clearly:

```
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
```

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., Multi-dimensional Arrays).

It is sometimes convenient to terminate the repetition of a `while` before the test condition is falsified or stop iterating in a `for` loop before the end of the iterable object is reached. This can be accomplished with the `break` keyword:

```
julia> i = 1;
```

```julia
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
```

Without the `break` keyword, the above `while` loop would never terminate on its own, and the `for` loop would iterate up to 1000. These loops are both exited early by using `break`.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The `continue` keyword accomplishes this:

```julia
julia> for i = 1:10
           if i % 3 != 0
               continue
           end
           println(i)
       end
3
6
9
```

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the `println` call inside the `if` block. In realistic usage there is more code to be evaluated after the `continue`, and often there are multiple points from which one calls `continue`.

Multiple nested `for` loops can be combined into a single outer loop, forming the cartesian product of its iterables:

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

With this syntax, iterables may still refer to outer loop variables; e.g. `for i = 1:n, j = 1:i` is valid. However a `break` statement inside such a loop exits the entire nest of loops, not just the inner one. Both variables (`i` and `j`) are set to their current iteration values each time the inner loop runs. Therefore, assignments to `i` will not be visible to subsequent iterations:

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

If this example were rewritten to use a `for` keyword for each variable, then the output would be different: the second and fourth values would contain `0`.

8.5   Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program while printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances then allow that code to take the appropriate action.

Built-in `Exceptions`

`Exceptions` are thrown when an unexpected condition has occurred. The built-in `Exceptions` listed below all interrupt the normal flow of control.

For example, the `sqrt` function throws a `DomainError` if applied to a negative real value:

```julia
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:
 [...]
```

You may define your own exceptions in the following way:

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

The `throw` function

Exceptions can be created explicitly with `throw`. For example, a function defined only for nonnegative numbers could be written to `throw` a `DomainError` if the argument is negative:

```julia
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
```

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

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

Note that DomainError without parentheses is not an exception, but a type of exception. It needs to be called to obtain an Exception object:

```julia
julia> typeof(DomainError(nothing)) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

Additionally, some exception types take one or more arguments that are used for error reporting:

```julia
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: x not defined
```

This mechanism can be implemented easily by custom exception types following the way UndefVarError is written:

```julia
julia> struct MyUndefVarError <: Exception
           var::Symbol
```

```
        end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

> **Note**
>
> When writing an error message, it is preferred to make the first word lowercase. For example, `size(A)`
> `== size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`
>
> is preferred over
>
> `size(A) == size(B) || throw(DimensionMismatch("Size of A not equal to size of B"))`.
>
> However, sometimes it makes sense to keep the uppercase first letter, for instance if an argument to
> a function is a capital letter: `size(A,1) == size(B,2) || throw(DimensionMismatch("A has`
> `first dimension..."))`.

Errors

The `error` function is used to produce an `ErrorException` that interrupts the normal flow of control.

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the `sqrt` function that raises an error if its argument is negative:

```
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
```

If `fussy_sqrt` is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

```
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
```

The `try/catch` statement

The `try/catch` statement allows for `Exceptions` to be tested for. For example, a customized square root function can be written to automatically call either the real or complex square root method on demand using `Exceptions`:

```
julia> f(x) = try
           sqrt(x)
       catch
           sqrt(complex(x, 0))
       end
f (generic function with 1 method)

julia> f(1)
1.0

julia> f(-1)
0.0 + 1.0im
```

It is important to note that in real code computing this function, one would compare x to zero instead of catching an exception. The exception is much slower than simply comparing and branching.

`try/catch` statements also allow the `Exception` to be saved in a variable. The following contrived example calculates the square root of the second element of x if x is indexable, otherwise assumes x is a real number and returns its square root:

```
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:
[...]
```

Note that the symbol following `catch` will always be interpreted as a name for the exception, so care is needed when writing `try/catch` expressions on a single line. The following code will *not* work to return the value of `x` in case of an error:

```
try bad() catch x end
```

Instead, use a semicolon or insert a line break after `catch`:

```
try bad() catch; x end

try bad()
catch
    x
end
```

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. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. Julia provides the `rethrow`, `backtrace` and `catch_backtrace` functions for more advanced error handling.

`finally` **Clauses**

In code that performs state changes or uses resources like files, there is typically clean-up work (such as closing files) that needs to be done when the code is finished. Exceptions potentially complicate this task, since they can cause a block of code to exit before reaching its normal end. The `finally` keyword provides a way to 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.

8.6   Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. This feature is sometimes called by 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.

This kind of control flow can make it much easier to solve certain problems. 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
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
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 which creates a channel bound task 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:

```julia
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.

Note that currently Julia tasks are not scheduled to run on separate CPU cores. True kernel threads are discussed under the topic of Parallel Computing.

Core task operations

Let us explore the low level construct `yieldto` to understand how task switching works. `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.

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. It is also possible to make the scheduler run a task whenever it can, without necessarily waiting for any events. This is done by calling `schedule`, or using the `@async` macro (see Parallel Computing for more details).

Task states

Tasks have a `state` field that describes their execution status. A `Task` `state` is one of the following symbols:

Symbol	Meaning
`:runnable`	Currently running, or available to be switched to
`:waiting`	Blocked waiting for a specific event
`:queued`	In the scheduler's run queue about to be restarted
`:done`	Successfully finished executing
`:failed`	Finished with an uncaught exception

Chapter 9

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:

Chapter 10

- Scope blocks that may nest only in other global scope blocks:

    - global scope

        * module, baremodule
        * at interactive prompt (REPL)

    - local scope (don't allow nesting)

        * (mutable) struct, macro

- Scope blocks which may nest anywhere (in global or local scope):

    - local scope

        * for, while, try-catch-finally, let
        * functions (either syntax, anonymous & do-blocks)
        * comprehensions, broadcast-fusing

Notably missing from this table are begin blocks and if blocks which do *not* introduce new scope blocks. 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
julia> module Bar
           x = 1
           foo() = x
       end;
```

and not a x in the scope where foo is used:

```julia
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
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. Additionally, the local scope inherits all globals that are assigned to in its parent global scope block (if it is surrounded by a global `if` or `begin` scope). 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 does not back-propagate to its parent scope. For example, here the z is not introduced into the top-level scope:

```julia
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.)

Inside a local scope a variable can be forced to be a new local variable using the `local` keyword:

```julia
julia> x = 0;

julia> for i = 1:10
           local x # this is also the default
           x = i + 1
       end

julia> x
0
```

Inside a local scope a global variable can be assigned to by using the keyword `global`:

```julia
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
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.

Local scopes are introduced by most block keywords, with notable exceptions of `begin` and `if`.

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 but not for writing:

```julia
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. One reason for this is that remotely changing the state of global variables in other modules should be done with care as it makes the local behavior of the program hard 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 a 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 vs. 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
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
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
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
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
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
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
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 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 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
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
julia> const z = 100
100

julia> z = 100
100
```

The last rule applies for immutable objects even if the vairable binding would change, e.g.:

```julia
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
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 possible, changing the value of a variable that is declared as constant is strongly discouraged. 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
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 Methods, 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 `structs` 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
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
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
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:

```julia
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 Methods 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:

```julia
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:

```julia
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:

```julia
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 Methods 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
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
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
julia> Foo((), 23.5, 1)
ERROR: InexactError: Int64(Int64, 23.5)
Stacktrace:
[...]
```

You may find a list of field names using the `fieldnames` function.

```julia
julia> fieldnames(Foo)
(:bar, :baz, :qux)
```

You can access the field values of a composite object using the traditional `foo.bar` notation:

```julia
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
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 Methods, 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
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
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
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
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
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
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

Other types, of course, are not subtypes of it:

```julia
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Concrete Point types with different values of T are never subtypes of each other:

```julia
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 Methods.

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
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
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
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
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

Parametric abstract types are invariant, much as parametric composite types are:

```julia
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
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
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
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{AbstractString} <: Pointy{AbstractString}
true
```

This relationship is also invariant:

```julia
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
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, Methods.

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
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
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:

```julia
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:

```julia
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:

```julia
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
julia> typeof((1,"foo",2.5))
Tuple{Int64,String,Float64}
```

Note the implications of covariance:

```julia
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
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 Varargs Functions).

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
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
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
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
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
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:

```julia
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:

```julia
# 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, the floating point register sizes are specified by the IEEE-754 standard. Whereas the size of `Int` reflects the size of a native pointer on that machine.)

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,Float64,Rational})
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
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
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
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
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
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
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
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
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
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
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))
3.04.0im

julia> [Polar(3, 4.0) Polar(4.0,5.3)]
1×2 Array{Polar{Float64},2}:
 3.04.0im  4.05.3im
```

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
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
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

Methods

Recall from Functions that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the + function.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument. [1] Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as multiple dispatch. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in x + y belong to x any more than it does to y? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a powerful and convenient paradigm for structuring and organizing programs.

12.1 Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions

---

[1] In C++ or Java, for example, in a method call like `obj.meth(arg1,arg2)`, the object obj "receives" the method call and is implicitly passed to the method via the `this` keyword, rather than as an explicit method argument. When the current `this` object is the receiver of a method call, it can be omitted altogether, writing just `meth(arg1,arg2)`, with `this` implied as the receiving object.

and operators, like the aforementioned + function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the :: type-assertion operator, introduced in the section on Composite Types:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

This function definition applies only to calls where x and y are both values of type Float64:

```
julia> f(2.0, 3.0)
7.0
```

Applying it to any other types of arguments will result in a MethodError:

```
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)
```

As you can see, the arguments must be precisely of type Float64. Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because Float64 is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type Float64. It may often be useful, however, to write more general methods where the declared parameter types are abstract:

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

This method definition applies to any pair of arguments that are instances of Number. They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression 2x - y.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method

definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for f over all pairs of instances of the abstract type Number – but with a different behavior specific to pairs of Float64 values. If one of the arguments is a 64-bit float but the other one is not, then the f(Float64,Float64) method cannot be called and the more general f(Number,Number) method must be used:

```
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
```

The 2x + y definition is only used in the first case, while the 2x - y definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. Conversion and Promotion, however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic. [2]

For non-numeric values, and for fewer or more than two arguments, the function f remains undefined, and applying it will still result in a MethodError:

```
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
```

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

```
julia> f
f (generic function with 2 methods)
```

This output tells us that f is a function object with two methods. To find out what the signatures of those methods are, use the methods function:

```
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
```

which shows that f has two methods, one taking two Float64 arguments and one taking arguments of type Number. It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number none:1.

In the absence of a type declaration with ::, the type of a method parameter is Any by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type Any. Thus, we can define a catch-all method for f like so:

```julia
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> f("foo", 1)
Whoa there, Nelly.
```

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it will only be called on pairs of arguments to which no other method definition applies.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:

```julia
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
```

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

12.2   Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:

```julia
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)
```

Here the call g(2.0, 3.0) could be handled by either the g(Float64, Any) or the g(Any, Float64) method, and neither is more specific than the other. In such cases, Julia raises a `MethodError` rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case:

```
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
```

It is recommended that the disambiguating method be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

In more complex cases, resolving method ambiguities involves a certain element of design; this topic is explored further below.

12.3 Parametric Methods

Method definitions can optionally have type parameters qualifying the signature:

```
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)
```

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

```
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
```

Such definitions correspond to methods whose type signatures are UnionAll types (see UnionAll Types).

This kind of definition of function behavior by dispatch is quite common – idiomatic, even – in Julia. Method type parameters are not restricted to being used as the types of arguments: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter T is used as the type parameter to the parametric type Vector{T} in the method signature:

```
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
```

As you can see, the type of the appended element must match the element type of the vector it is appended to, or else a MethodError is raised. In the following example, the method type parameter T is used as the return value:

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)
```

```
julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

Just as you can put subtype constraints on type parameters in type declarations (see Parametric Types), you can also constrain type parameters of methods:

```
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<:Number, ::T<:Number) 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
```

The `same_type_numeric` function behaves much like the `same_type` function defined above, but is only defined for pairs of numbers.

Parametric methods allow the same syntax as `where` expressions used to write types (see UnionAll Types). If there is only a single parameter, the enclosing curly braces (in `where {T}`) can be omitted, but are often preferred for clarity. Multiple parameters can be separated with commas, e.g. `where {T, S<:Real}`, or written using nested `where`, e.g. `where S<:Real where T`.

12.4 Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to Julia's ability to statically infer and compile code to run fast, without the usual JIT tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including Tasks and Threads (and any previously defined `@generated` functions). Let's start with an example to see what this means:

```
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
```

In this example, observe that the new definition for newfun has been created, but can't be immediately called. The new global is immediately visible to the tryeval function, so you could write return newfun (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to newfun *from the REPL* work as expected, being able to both see and call the new definition of newfun.

However, future calls to tryeval will continue to see the definition of newfun as it was *at the previous statement at the REPL*, and thus before that call to tryeval.

You may want to try this for yourself to see how it works.

The implementation of this behavior is a "world age counter". This monotonically increasing value tracks each method definition operation. This allows describing "the set of method definitions visible to a given runtime environment" as a single number, or "world age". It also allows comparing the methods available in two worlds just by comparing their ordinal value. In the example above, we see that the "current world" (in which the method newfun exists), is one greater than the task-local "runtime world" that was fixed when the execution of tryeval started.

Sometimes it is necessary to get around this (for example, if you are implementing the above REPL). Fortunately, there is an easy solution: call the function using Base.invokelatest:

```
julia> function tryeval2()
           @eval newfun2() = 2
           Base.invokelatest(newfun2)
       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

Finally, let's take a look at some more complex examples where this rule comes into play. Define a function f(x), which initially has one method:

```
julia> f(x) = "original definition"
f (generic function with 1 method)
```

Start some other operations that use f(x):

```
julia> g(x) = f(x)
g (generic function with 1 method)
```

```julia
julia> t = @async f(wait()); yield();
```

Now we add some new methods to `f(x)`:

```julia
julia> f(x::Int) = "definition for Int"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "definition for Type{Int}"
f (generic function with 3 methods)
```

Compare how these results differ:

```julia
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> fetch(schedule(t, 1))
"original definition"

julia> t = @async f(wait()); yield();

julia> fetch(schedule(t, 1))
"definition for Int"
```

12.5   Design Patterns with Parametric Methods

While complex dispatch logic is not required for performance or usability, sometimes it can be the best way to express some algorithm. Here are a few common design patterns that come up sometimes when using dispatch in this way.

Extracting the type parameter from a super-type

Here is the correct code template for returning the element-type T of any arbitrary subtype of `AbstractArray`:

```julia
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

using so-called triangular dispatch. Note that if T is a `UnionAll` type, as e.g. `eltype(Array{T} where T <: Integer)`, then Any is returned (as does the the version of `eltype` in Base).

Another way, which used to be the only correct way before the advent of triangular dispatch in Julia v0.6, is:

```julia
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))
```

Another possibility is the following, which could useful to adapt to cases where the parameter T would need to be matched more narrowly:

```
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))
```

One common mistake is to try and get the element-type by using introspection:

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

However, it is not hard to construct cases where this will fail:

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

Here we have created a type `BitVector` which has no parameters, but where the element-type is still fully specified, with T equal to `Bool`!

Building a similar type with a different type parameter

When building generic code, there is often a need for constructing a similar object with some change made to the layout of the type, also necessitating a change of the type parameters. For instance, you might have some sort of abstract array with an arbitrary element type and want to write your computation on it with a specific element type. We must implement a method for each `AbstractArray{T}` subtype that describes how to compute this type transform. There is no general transform of one subtype into another subtype with a different parameter. (Quick review: do you see why this is?)

The subtypes of `AbstractArray` typically implement two methods to achieve this: A method to convert the input array to a subtype of a specific `AbstractArray{T, N}` abstract type; and a method to make a new uninitialized array with a specific element type. Sample implementations of these can be found in Julia Base. Here is a basic example usage of them, guaranteeing that `input` and `output` are of the same type:

```
input = convert(AbstractArray{Eltype}, input)
output = similar(input, Eltype)
```

As an extension of this, in cases where the algorithm needs a copy of the input array, `convert` is insufficient as the return value may alias the original input. Combining `similar` (to make the output array) and `copyto!` (to fill it with the input data) is a generic way to express the requirement for a mutable copy of the input argument:

```
copy_with_eltype(input, Eltype) = copyto!(similar(input, Eltype), input)
```

Iterated dispatch

In order to dispatch a multi-level parametric argument list, often it is best to separate each level of dispatch into distinct functions. This may sound similar in approach to single-dispatch, but as we shall see below, it is still more flexible.

For example, trying to dispatch on the element-type of an array will often run into ambiguous situations. Instead, commonly code will dispatch first on the container type, then recurse down to a more specific method based on eltype. In most cases, the algorithms lend themselves conveniently to this hierarchical approach, while in other cases, this rigor must be resolved manually. This dispatching branching can be observed, for example, in the logic to sum two matrices:

```
# First dispatch selects the map algorithm for element-wise summation.
+(a::Matrix, b::Matrix) = map(+, a, b)
# Then dispatch handles each element and selects the appropriate
# common element type for the computation.
+(a, b) = +(promote(a, b)...)
# Once the elements have the same type, they can be added.
# For example, via primitive operations exposed by the processor.
+(a::Float64, b::Float64) = Core.add(a, b)
```

Trait-based dispatch

A natural extension to the iterated dispatch above is to add a layer to method selection that allows to dispatch on sets of types which are independent from the sets defined by the type hierarchy. We could construct such a set by writing out a Union of the types in question, but then this set would not be extensible as Union-types cannot be altered after creation. However, such an extensible set can be programmed with a design pattern often referred to as a "Holy-trait".

This pattern is implemented by defining a generic function which computes a different singleton value (or type) for each trait-set to which the function arguments may belong to. If this function is pure there is no impact on performance compared to normal dispatch.

The example in the previous section glossed over the implementation details of map and promote, which both operate in terms of these traits. When iterating over a matrix, such as in the implementation of map, one important question is what order to use to traverse the data. When AbstractArray subtypes implement the Base.IndexStyle trait, other functions such as map can dispatch on this information to pick the best algorithm (see Abstract Array Interface). This means that each subtype does not need to implement a custom version of map, since the generic definitions + trait classes will enable the system to select the fastest version. Here a toy implementation of map illustrating the trait-based dispatch:

```
map(f, a::AbstractArray, b::AbstractArray) = map(Base.IndexStyle(a, b), f, a, b)
# generic implementation:
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray) = ...
# linear-indexing implementation (faster)
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) = ...
```

This trait-based approach is also present in the promote mechanism employed by the scalar +. It uses promote_type, which returns the optimal common type to compute the operation given the two types of the operands. This makes it possible to reduce the problem of implementing every function for every pair of possible type arguments, to the much smaller problem of implementing a conversion operation from each type to a common type, plus a table of preferred pair-wise promotion rules.

Output-type computation

The discussion of trait-based promotion provides a transition into our next design pattern: computing the output element type for a matrix operation.

For implementing primitive operations, such as addition, we use the promote_type function to compute the desired output type. (As before, we saw this at work in the promote call in the call to +).

For more complex functions on matrices, it may be necessary to compute the expected return type for a more complex sequence of operations. This is often performed by the following steps:

1. Write a small function op that expresses the set of operations performed by the kernel of the algorithm.

2. Compute the element type R of the result matrix as promote_op(op, argument_types...), where argument_types is computed from eltype applied to each input array.

3. Build the output matrix as `similar(R, dims)`, where `dims` are the desired dimensions of the output array.

For a more specific example, a generic square-matrix multiply pseudo-code might look like:

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## this is insufficient because it assumes `one(eltype(a))` is constructable:
    # R = typeof(op(one(eltype(a)), one(eltype(b))))

    ## this fails because it assumes `a[1]` exists and is representative of all elements of the
    ↪   array
    # R = typeof(op(a[1], b[1]))

    ## this is incorrect because it assumes that `+` calls `promote_type`
    ## but this is not true for some types, such as Bool:
    # R = promote_type(ai, bi)

    # this is wrong, since depending on the return value
    # of type-inference is very brittle (as well as not being optimizable):
    # R = Base.return_types(op, (eltype(a), eltype(b)))

    ## but, finally, this works:
    R = promote_op(op, eltype(a), eltype(b))
    ## although sometimes it may give a larger type than desired
    ## it will always give a correct type

    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(b, 1)
                ## here we don't use `ab = zero(R)`,
                ## since `R` might be `Any` and `zero(Any)` is not defined
                ## we also must declare `ab::R` to make the type of `ab` constant in the loop,
                ## since it is possible that typeof(a * b) != typeof(a * b + a * b) == R
                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
```

Separate convert and kernel logic

One way to significantly cut down on compile-times and testing complexity is to isolate the logic for converting to the desired type and the computation. This lets the compiler specialize and inline the conversion logic independent from the rest of the body of the larger kernel.

This is a common pattern seen when converting from a larger class of types to the one specific argument type that is actually supported by the algorithm:

```
complexfunction(arg::Int) = ...
complexfunction(arg::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

12.6   Parametrically-constrained Varargs methods

Function parameters can also be used to constrain the number of arguments that may be supplied to a "varargs" function (Varargs Functions). The notation `Vararg{T,N}` is used to indicate such a constraint. For example:

```
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
```

More usefully, it is possible to constrain varargs methods by a parameter. For example:

```
function getindex(A::AbstractArray{T,N}, indices::Vararg{Number,N}) where {T,N}
```

would be called only when the number of `indices` matches the dimensionality of the array.

When only the type of supplied arguments needs to be constrained `Vararg{T}` can be equivalently written as `T....` For instance `f(x::Int...) = x` is a shorthand for `f(x::Vararg{Int}) = x`.

12.7   Note on Optional and keyword Arguments

As mentioned briefly in Functions, optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

```
f(a=1,b=2) = a+2b
```

translates to the following three methods:

```
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:

```
f(a::Int,b::Int) = a-2b
```

then the result of both f() and f(1,2) is -3. In other words, optional arguments are tied to a function, not to any specific method of that function. It depends on the types of the optional arguments which method is invoked. When optional arguments are defined in terms of a global variable, the type of the optional argument may even change at run-time.

Keyword arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with keyword arguments processed after the matching method is identified.

12.8   Function-like objects

Methods are associated with types, so it is possible to make any arbitrary Julia object "callable" by adding methods to its type. (Such "callable" objects are sometimes called "functors.")

For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

```
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)
```

Notice that the function is specified by type instead of by name. As with normal functions there is a terse syntax form. In the function body, p will refer to the object that was called. A Polynomial can be used as follows:

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> p(3)
931

julia> p()
2551
```

This mechanism is also the key to how type constructors and closures (inner functions that refer to their surrounding environment) work in Julia.

12.9   Empty generic functions

Occasionally it is useful to introduce a generic function without yet adding methods. This can be used to separate interface definitions from implementations. It might also be done for the purpose of documentation or code readability. The syntax for this is an empty function block without a tuple of arguments:

```
function emptyfunc
end
```

12.10 Method design and the avoidance of ambiguities

Julia's method polymorphism is one of its most powerful features, yet exploiting this power can pose design challenges. In particular, in more complex method hierarchies it is not uncommon for ambiguities to arise.

Above, it was pointed out that one can resolve ambiguities like

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

by defining a method

```
f(x::Int, y::Int) = 3
```

This is often the right strategy; however, there are circumstances where following this advice blindly can be counter-productive. In particular, the more methods a generic function has, the more possibilities there are for ambiguities. When your method hierarchies get more complicated than this simple example, it can be worth your while to think carefully about alternative strategies.

Below we discuss particular challenges and some alternative ways to resolve such issues.

Tuple and NTuple arguments

`Tuple` (and `NTuple`) arguments present special challenges. For example,

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

are ambiguous because of the possibility that `N == 0`: there are no elements to determine whether the `Int` or `Float64` variant should be called. To resolve the ambiguity, one approach is define a method for the empty tuple:

```
f(x::Tuple{}) = 3
```

Alternatively, for all methods but one you can insist that there is at least one element in the tuple:

```
f(x::NTuple{N,Int}) where {N} = 1          # this is the fallback
f(x::Tuple{Float64, Vararg{Float64}}) = 2  # this requires at least one Float64
```

Orthogonalize your design

When you might be tempted to dispatch on two or more arguments, consider whether a "wrapper" function might make for a simpler design. For example, instead of writing multiple variants:

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

you might consider defining

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

where g converts the argument to type A. This is a very specific example of the more general principle of orthogonal design, in which separate concepts are assigned to separate methods. Here, g will most likely need a fallback definition

```
g(x::A) = x
```

A related strategy exploits `promote` to bring x and y to a common type:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

One risk with this design is the possibility that if there is no suitable promotion method converting x and y to the same type, the second method will recurse on itself infinitely and trigger a stack overflow. The non-exported function `Base.promote_noncircular` can be used as an alternative; when promotion fails it will still throw an error, but one that fails faster with a more specific error message.

Dispatch on one argument at a time

If you need to dispatch on multiple arguments, and there are many fallbacks with too many combinations to make it practical to define all possible variants, then consider introducing a "name cascade" where (for example) you dispatch on the first argument and then call an internal method:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

Then the internal methods _fA and _fB can dispatch on y without concern about ambiguities with each other with respect to x.

Be aware that this strategy has at least one major disadvantage: in many cases, it is not possible for users to further customize the behavior of f by defining further specializations of your exported function f. Instead, they have to define specializations for your internal methods _fA and _fB, and this blurs the lines between exported and internal methods.

Abstract containers and element types

Where possible, try to avoid defining methods that dispatch on specific element types of abstract containers. For example,

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

generates ambiguities for anyone who defines a method

```
-(A::MyArrayType{T}, b::T) where {T}
```

The best approach is to avoid defining *either* of these methods: instead, rely on a generic method `-(A::AbstractArray, b)` and make sure this method is implemented with generic calls (like `similar` and `-`) that do the right thing for each container type and element type *separately*. This is just a more complex variant of the advice to orthogonalize your methods.

When this approach is not possible, it may be worth starting a discussion with other developers about resolving the ambiguity; just because one method was defined first does not necessarily mean that it can't be modified or eliminated. As a last resort, one developer can define the "band-aid" method

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

that resolves the ambiguity by brute force.

Complex method "cascades" with default arguments

If you are defining a method "cascade" that supplies defaults, be careful about dropping any arguments that correspond to potential defaults. For example, suppose you're writing a digital filtering algorithm and you have a method that handles the edges of the signal by applying padding:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # now perform the "real" computation
end
```

This will run afoul of a method that supplies default padding:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by default
```

Together, these two methods generate an infinite recursion with A constantly growing bigger.

The better design would be to define your call hierarchy like this:

```
struct NoPad end  # indicate that no padding is desired, or that it's already applied

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # default boundary conditions

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # indicate the new boundary conditions
end

# other padding methods go here

function myfilter(A, kernel, ::NoPad)
    # Here's the "real" implementation of the core computation
end
```

NoPad is supplied in the same argument position as any other kind of padding, so it keeps the dispatch hierarchy well organized and with reduced likelihood of ambiguities. Moreover, it extends the "public" `myfilter` interface: a user who wants to control the padding explicitly can call the NoPad variant directly.

---

[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
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. There are, however, cases where 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.

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:

---

[1]Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from 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.

```julia
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
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 much like an outer constructor method, with two differences:

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
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
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, but messing around with an object's internals uninvited is considered poor form. 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
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
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 considered good form 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 con-
structor method can then use the incomplete object, finishing its initialization before returning it. Here, for example,
we take another crack at defining the SelfReferential type, with 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, incompletely initialized
objects can be returned:

```
julia> mutable struct Incomplete
           xx
           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
julia> z.xx
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
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
julia> mutable struct Lazy
           xx
           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 `xx` 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
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<:Real, ::T<:Real) 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(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
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
Closest candidates are:
  Point(::T<:Real, !Matched::T<:Real) 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
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
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
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 Rational Numbers – 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 do the core work of making an instance. 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 Integers and Floating-Point Numbers, Mathematical Operations and Elementary Functions, 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
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
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{UInt8}, ::Type{Int8}) = Int
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

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*)

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.

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:

```julia
for i in iter   # or  "for i = iter"
    # body
end
```

is translated into:

```julia
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
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
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
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
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
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
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
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
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
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 Ints, separate methods must be written:

```julia
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.

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::NTuple{Int})`	`similar(A, eltype(A), dims)`	Return a mutable array with the same element type and size *dims*
`similar(A, ::Type{S}, dims::NTuple{Int})`	`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
`Base.similar(A, ::Type{S}, inds::NTuple{Ind})`	`similar(A, S, Base.to_shape(inds))`	Return a mutable array with the specified indices inds (see below)
`Base.similar(T::Union{Type, Function}, inds)`	`T(Base.to_shape(inds))`	Return an array similar to T with the specified indices inds (see below)

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
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
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
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

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)[k]`	Return the distance in memory (in number of elements) between adjacent elements in dimension k.

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.
```

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.broadcast_axes(x)`	Declaration of the indices of x for broadcasting purposes (defaults to `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

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 `AbstractArrays` and `Numbers` — 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 spe-
cific 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.

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:

```julia
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`, but only operates on a single name at a time. 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: Main, Core, and Base.

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.

Core 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.

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.

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.

Module file paths

The global variable `LOAD_PATH` contains the directories Julia searches for modules when calling `require`. It can be extended using `push!`:

```
push!(LOAD_PATH, "/Path/To/My/Module/")
```

Putting this statement in the file `~/.julia/config/startup.jl` will extend `LOAD_PATH` on every Julia startup. Alternatively, the module load path can be extended by defining the environment variable JULIA_LOAD_PATH.

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 want to have changes to the source reflected in the running system, you should call `reload("Module")` on the module you changed, and any module that depended on it in which you want to see the change reflected.

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.

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.

   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 it 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. By default, documentation is expected to be written in Markdown, and the doc"" string macro simply creates an object representing the Markdown content. In the future it is likely to do more advanced things such as allowing for relative image or link paths.

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 easy to use an arbitrary object (here a `raw` string) 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.

`doc""` should only be used when the docstring contains $ or \ characters that should not be parsed by Julia such as LaTeX syntax or Julia source code examples containing interpolation.

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 `Method`s, 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 Module M. 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 `baremodules` cannot be documented.

Global Variables

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring "..." to the `Bindings` a, b, and c.

`Bindings` 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 Enums. 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

17.5   Markdown syntax

The following markdown syntax is supported in Julia.

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 *emphasised* 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 LaTeXmarkup then two enclosing backticks is sufficient.

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:

```
"""
    accumulate!(op, y, x)

Cumulative operation `op` on a vector `x`, storing the result in `y`. See also
↪  [`accumulate`](@ref).
"""
```

This will create a link in the generated docs to the `accumulate` documentation (which has more information about what this function actually does). 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.

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 emphasised 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.

17.6 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 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
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
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
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
julia> :foo == Symbol("foo")
true

julia> Symbol("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

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> :(:)
:(:)

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
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
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`. The parser yields `QuoteNodes` 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
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
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
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
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
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, 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
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
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
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:

```julia
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
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 four 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 – and any function that was defined *before* the definition of the generated function.

3. Instead of calculating something or performing some action, you return a *quoted expression* which, when evaluated, does what you want.

4. 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
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
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
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
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
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
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
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:

```julia
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

Multi-dimensional Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia. As such, it's also possible to define custom array types by inheriting from `AbstractArray`. See the manual section on the AbstractArray interface for more details on implementing a custom array type.

An array is a collection of objects stored in a multi-dimensional grid. In the most general case, an array may contain objects of type Any. For most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`.

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions are passed by sharing (i.e. by pointers). Some technical computing languages pass arrays by value, and while this prevents accidental modification by callees of a value in the caller, it makes avoiding unwanted copying of arrays difficult. By convention, a function name ending with a `!` indicates that it will mutate or destroy the value of one or more of its arguments (see, for example, `sort` and `sort!`. Callees must make explicit copies to ensure that they don't modify inputs that they don't intend to change. Many non- mutating functions are implemented by calling a function of the same name with an added `!` at the end on an explicit copy of the input, and returning that copy.

19.1   Basic Functions

Function	Description
`eltype(A)`	the type of the elements contained in A
`length(A)`	the number of elements in A
`ndims(A)`	the number of dimensions of A
`size(A)`	a tuple containing the dimensions of A
`size(A,n)`	the size of A along dimension n
`axes(A)`	a tuple containing the valid indices of A
`axes(A,n)`	a range expressing the valid indices along dimension n
`eachindex(A)`	an efficient iterator for visiting each position in A
`stride(A,k)`	the stride (linear index distance between adjacent elements) along dimension k
`strides(A)`	a tuple of the strides in each dimension

19.2   Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a `dims...` argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments. Most of these functions also accept a first input T, which is the element type of the array. If the type T is omitted it will default to `Float64`.

Function	Description
`Array{T}(undef, dims...)`	an uninitialized dense `Array`
`zeros(T, dims...)`	an `Array` of all zeros
`ones(T, dims...)`	an `Array` of all ones
`trues(dims...)`	a `BitArray` with all values `true`
`falses(dims...)`	a `BitArray` with all values `false`
`reshape(A, dims...)`	an array containing the same data as A, but with different dimensions
`copy(A)`	copy A
`deepcopy(A)`	copy A, recursively copying its elements
`similar(A, T, dims...)`	an uninitialized array of the same type as A (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of A if omitted.
`reinterpret(T, A)`	an array with the same binary data as A, but with element type T
`rand(T, dims...)`	an `Array` with random, iid [1] and uniformly distributed values in the half-open interval $[0, 1)$
`randn(T, dims...)`	an `Array` with random, iid and standard normally distributed values
`Matrix{T}(I, m, n)`	m-by-n identity matrix
`range(start, stop=stop, length=n)`	range of n linearly spaced elements from `start` to `stop`
`fill!(A, x)`	fill the array A with the value x
`fill(x, dims...)`	an `Array` filled with the value x

The syntax `[A, B, C, ...]` constructs a 1-d array (vector) of its arguments. If all arguments have a common promotion type then they get converted to that type using `convert`.

To see the various ways we can pass dimensions to these constructors, consider the following examples:

```julia
julia> zeros(Int8, 2, 2)
2×2 Array{Int8,2}:
 0  0
 0  0

julia> zeros(Int8, (2, 2))
2×2 Array{Int8,2}:
 0  0
 0  0
```

---

[1] *iid*, independently and identically distributed.

```
julia> zeros((2, 2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

Here, (2, 2) is a Tuple.

19.3 Concatenation

Arrays can be constructed and also concatenated using the following functions:

Function	Description
cat(A...; dims=k)	concatenate input arrays along dimension(s) k
vcat(A...)	shorthand for cat(A...; dims=1)
hcat(A...)	shorthand for cat(A...; dims=2)

Scalar values passed to these functions are treated as 1-element arrays. For example,

```
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
```

The concatenation functions are used so often that they have special syntax:

Expression	Calls
[A; B; C; ...]	vcat
[A B C ...]	hcat
[A B; C D; ...]	hvcat

hvcat concatenates in both dimension 1 (with semicolons) and dimension 2 (with spaces). Consider these examples of this syntax:

```
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   Typed array initializers

An array with a specific element type can be constructed using the syntax `T[A, B, C, ...]`. This will construct a 1-d array with element type T, initialized to contain elements A, B, C, etc. For example, `Any[x, y, z]` constructs a heterogeneous array that can contain any values.

Concatenation syntax can similarly be prefixed with a type to specify the element type of the result.

```
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   Comprehensions

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

The meaning of this form is that `F(x,y,...)` is evaluated with the variables x, y, etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like `1:n` or `2:(n-1)`, or explicit arrays of values like `[1.2, 3.4, 5.7]`. The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges rx, ry, etc. and each `F(x,y,...)` evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbor along a 1-d grid. :

```
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
```

The resulting array type depends on the types of the computed elements. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

19.6 Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance (see Iteration). For example, the following expression sums a series without allocating memory:

```julia
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

When writing a generator expression with multiple dimensions inside an argument list, parentheses are needed to separate the generator from subsequent arguments:

```julia
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

All comma-separated expressions after `for` are interpreted as ranges. Adding parentheses lets us add a third argument to map:

```julia
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. As in other cases of inner functions 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 described in performance tips.

Ranges in generators and comprehensions can depend on previous ranges by writing multiple `for` keywords:

```julia
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)
```

In such cases, the result is always 1-d.

Generated values can be filtered using the `if` keyword:

```julia
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   Indexing

The general syntax for indexing into an n-dimensional array A is:

```
X = A[I_1, I_2, ..., I_n]
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes Colon (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If all the indices are scalars, then the result X is a single element from the array A. Otherwise, X is an array with the same number of dimensions as the sum of the dimensionalities of all the indices.

If all indices are vectors, for example, then the shape of X would be (`length(I_1), length(I_2), ..., length(I_n)`), with location (`i_1, i_2, ..., i_n`) of X containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`.

Example:

```julia
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.

If I_1 is changed to a two-dimensional matrix, then X becomes an n+1-dimensional array of shape (size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n)). The matrix adds a dimension.

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
```

The location (i_1, i_2, i_3, ..., i_{n+1}) contains the value at A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]. All dimensions indexed with scalars are dropped. For example, the result of A[2, I, 3] is an array with size size(I). Its ith element is populated by A[2, I[i], 3].

As a special part of this syntax, the end keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the innermost array being indexed. Indexing syntax without the end keyword is equivalent to a call to getindex:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

Example:

```
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
```

Empty ranges of the form n:n-1 are sometimes used to indicate the inter-index location between n-1 and n. For example, the searchsorted function uses this convention to indicate the insertion point of a value not found in a sorted array:

```
julia> a = [1,2,5,6,7];

julia> searchsorted(a, 3)
3:2
```

19.8   Assignment

The general syntax for assigning values in an n-dimensional array A is:

```
A[I_1, I_2, ..., I_n] = X
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes Colon (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If X is an array, it must have the same number of elements as the product of the lengths of the indices: `prod(length(I_1)`, `length(I_2), ..., length(I_n))`. The value in location `I_1[i_1], I_2[i_2], ..., I_n[i_n]` of A is over-written with the value `X[i_1, i_2, ..., i_n]`. If X is not an array, its value is written to all referenced locations of A.

Just as in Indexing, the end keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the array being assigned into. Indexed assignment syntax without the end keyword is equivalent to a call to `setindex!`:

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

Example:

```
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   Supported index types

In the expression `A[I_1, I_2, ..., I_n]`, each `I_k` may be a scalar index, an array of scalar indices, or an object that represents an array of scalar indices and can be converted to such by `to_indices`:

1. A scalar index. By default this includes:

    – Non-boolean integers

    – `CartesianIndex{N}`s, which behave like an N-tuple of integers spanning multiple dimensions (see below for more details)

2. An array of scalar indices. This includes:

    – Vectors and multidimensional arrays of integers

    – Empty arrays like [ ], which select no elements

- Ranges like a:c or a:b:c, which select contiguous or strided subsections from a to c (inclusive)

- Any custom array of scalar indices that is a subtype of `AbstractArray`

- Arrays of `CartesianIndex{N}` (see below for more details)

3. An object that represents an array of scalar indices and can be converted to such by `to_indices`. By default this includes:

- `Colon()` (:), which represents all indices within an entire dimension or across the entire array

- Arrays of booleans, which select elements at their `true` indices (see below for more details)

Some examples:

```julia
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
```

Cartesian indices

The special `CartesianIndex{N}` object represents a scalar index that behaves like an N-tuple of integers spanning multiple dimensions. For example:

```julia
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
```

Considered alone, this may seem relatively trivial; `CartesianIndex` simply gathers multiple integers together into one object that represents a single multidimensional index. When combined with other indexing forms and iterators that yield `CartesianIndexes`, however, this can lead directly to very elegant and efficient code. See Iteration below, and for some more advanced examples, see this blog post on multidimensional algorithms and iteration.

Arrays of `CartesianIndex{N}` are also supported. They represent a collection of scalar indices that each span N dimensions, enabling a form of indexing that is sometimes referred to as pointwise indexing. For example, it enables accessing the diagonal elements from the first "page" of A from above:

```julia
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
```

This can be expressed much more simply with dot broadcasting and by combining it with a normal integer index (instead of extracting the first page from A as a separate step). It can even be combined with a `:` to extract both diagonals from the two pages at the same time:

```julia
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
```

> **Warning**
>
> CartesianIndex and arrays of CartesianIndex are not compatible with the end keyword to rep-
> resent the last index of a dimension. Do not use end in indexing expressions that may contain either
> CartesianIndex or arrays thereof.

Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are true. Indexing by a boolean vector B is effectively the same as indexing by the vector of integers that is returned by findall(B). Similarly, indexing by a N-dimensional boolean array is effectively the same as indexing by the vector of CartesianIndex{N}s where its values are true. A logical index must be a vector of the same length as the dimension it indexes into, or it must be the only index provided and match the size and dimensionality of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling findall.

```julia
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}:
  true  false  false  false
  true  false  false  false
 false  false  false  false
  true   true  false   true

julia> x[mask]
5-element Array{Int64,1}:
  1
  2
  4
  8
 16
```

19.10   Iteration

The recommended ways to iterate over a whole array are

```julia
for a in A
    # Do something with the element a
end
```

```
for i in eachindex(A)
    # Do something with i and/or A[i]
end
```

The first construct is used when you need the value, but not index, of each element. In the second construct, `i` will be an `Int` if A is an array type with fast linear indexing; otherwise, it will be a `CartesianIndex`:

```
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)
```

In contrast with `for i = 1:length(A)`, iterating with `eachindex` provides an efficient way to iterate over any array type.

19.11   Array traits

If you write a custom `AbstractArray` type, you can specify that it has fast linear indexing using

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

This setting will cause `eachindex` iteration over a `MyArray` to use integers. If you don't specify this trait, the default value `IndexCartesian()` is used.

19.12   Array and Vectorized Operators and Functions

The following operators are supported for arrays:

1. Unary arithmetic – `-`, `+`

2. Binary arithmetic – `-`, `+`, `*`, `/`, `\`, `^`

3. Comparison – `==`, `!=`, `≈` (`isapprox`),

Most of the binary arithmetic operators listed above also operate elementwise when one argument is scalar: `-`, `+`, and `*` when either argument is scalar, and `/` and `\` when the denominator is scalar. For example, `[1, 2] + 3 == [4, 5]` and `[6, 4] / 2 == [3, 2]`.

Additionally, to enable convenient vectorization of mathematical and other operations, Julia provides the dot syntax `f.(args...)`, e.g. `sin.(x)` or `min.(x,y)`, for elementwise operations over arrays or mixtures of arrays and scalars (a Broadcasting operation); these have the additional advantage of "fusing" into a single loop when combined with other dot calls, e.g. `sin.(cos.(x))`.

Also, *every* binary operator supports a dot version that can be applied to arrays (and combinations of arrays and scalars) in such fused broadcasting operations, e.g. `z .== sin.(x .* y)`.

Note that comparisons such as `==` operate on whole arrays, giving a single boolean answer. Use dot operators like `.==` for elementwise comparisons. (For comparison operations like `<`, *only* the elementwise `.<` version is applicable to arrays.)

Also notice the difference between `max.(a,b)`, which broadcasts `max` elementwise over a and b, and `maximum(a)`, which finds the largest value within a. The same relationship holds for `min.(a,b)` and `minimum(a)`.

19.13 Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

```julia
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
```

This is wasteful when dimensions get large, so Julia offers broadcast, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given function elementwise:

```julia
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
```

Dotted operators such as `.+` and `.*` are equivalent to broadcast calls (except that they fuse, as described below). There is also a broadcast! function to specify an explicit destination (which can also be accessed in a fusing fashion by `.=` assignment). Moreover, `f.(args...)` is equivalent to `broadcast(f, args...)`, providing a convenient syntax to broadcast any function (dot syntax). Nested "dot calls" `f.(...)` (including calls to `.+` etcetera) automatically fuse into a single broadcast call.

Additionally, broadcast is not limited to arrays (see the function documentation), it also handles tuples and treats any argument that is not an array, tuple or Ref (except for Ptr) as a "scalar".

```julia
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   Implementation

The base array type in Julia is the abstract type `AbstractArray{T,N}`. It is parametrized by the number of dimensions N and the element type T. `AbstractVector` and `AbstractMatrix` are aliases for the 1-d and 2-d cases. Operations on `AbstractArray` objects are defined using higher level operators and functions, in a way that is independent of the underlying storage. These operations generally work correctly as a fallback for any specific array implementation.

The `AbstractArray` type includes anything vaguely array-like, and implementations of it might be quite different from conventional arrays. For example, elements might be computed on request rather than stored. However, any concrete `AbstractArray{T,N}` type should generally implement at least `size(A)` (returning an Int tuple), `getindex(A,i)` and `getindex(A,i1,...,iN)`; mutable arrays should also implement `setindex!`. It is recommended that these operations have nearly constant time complexity, or technically Õ(1) complexity, as otherwise some array functions may be unexpectedly slow. Concrete types should also typically provide a `similar(A,T=eltype(A),dims=size(A))` method, which is used to allocate a similar array for `copy` and other out-of-place operations. No matter how an `AbstractArray{T,N}` is represented internally, T is the type of object returned by *integer* indexing (A[1, ..., 1], when A is not empty) and N should be the length of the tuple returned by `size`. 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 `AbstractArrays`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. By putting the `@views` macro in front of an expression or block of code, any `array[...]` slice in that expression will be converted to create a `SubArray` view instead.

`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 kth 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 ap-

proporiate 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.

The following example computes the QR decomposition of a small section of a larger array, without creating any temporaries, and by calling the appropriate LAPACK function with the right leading dimension size and stride parameters.

```julia
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
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
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
julia> sum([1, missing])
missing
```

In this situation, use the `skipmissing` function to skip missing values

```julia
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> maximum(skipmissing([3, missing, 2, 1]))
3

julia> mean(skipmissing([3, missing, 2, 1]))
2.0

julia> mapreduce(sqrt, +, skipmissing([3, missing, 2, 1]))
4.146264369941973
```

Use `collect` to extract non-`missing` values and store them in an array

```
julia> collect(skipmissing([3, missing, 2, 1]))
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
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
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`:

```julia
for line in eachline(stdin)
    print("Found $line")
end
```

or `read` if you wanted to read by character instead:

```julia
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
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
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
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
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
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:

```julia
function read_and_capitalize(f::IOStream)
    return uppercase(read(f, String))
end
```

You can call:

```julia
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)
Base.TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
Base.TCPServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
Base.TCPServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
Base.TCPServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
Base.TCPServer(active)

julia> listen("testsocket") # Listens on a UNIX domain socket
Base.PipeServer(active)

julia> listen("\\\\.\\pipe\\testsocket") # Listens on a Windows named pipe
Base.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 to suspend and resume computations with full control of inter-`Tasks` communication without having to manually interface with the operative system's scheduler. Julia also allows to communicate between `Tasks` through operations like `wait` and `fetch`. Communication and data synchronization is managed through `Channel`s, which are the conduit that allows inter-`Tasks` communication.

Julia also supports experimental multi-threading, where execution is forked and an anonymous function is run across all threads. Described as a fork-join approach, parallel threads are branched off and they all have to join the Julia main thread to make serial execution continue. Multi-threading is supported using the `Base.Threads` module that is still considered experimental, as Julia is not fully thread-safe yet. In particular segfaults seem to emerge for I\O operations and task switching. As an un 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, so we will explain it later.

In the end we will present Julia's way to distributed and parallel computing. With scientific computing in mind, Julia natively implements interfaces to distribute a process through 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 Tasks in Control Flow discussed the execution of multiple functions in a co-operative manner. Channels 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 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
    @schedule 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
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
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
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
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 amout 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
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> @schedule make_jobs(n); # feed the jobs channel with "n" jobs
```

```julia
julia> for i in 1:4 # start 4 tasks to process requests in parallel
           @schedule 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,2)) seconds")
           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
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
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
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
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
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, m, fill(big(10)^20, nthreads()-1))]
            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)
```

271

```
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.

The syntax of remotecall is not especially convenient. The macro @spawn makes things easier. It operates on an expression rather than a function, and picks where to do the operation for you:

```
julia> r = @spawn rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawn 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, @spawn 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 @spawn 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 Futures have fetched, the remote stored value is deleted.

@async is similar to @spawn, 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 (dev v0.7) 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
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(@spawn 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:

```julia
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
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`. More-over, when `DummyModule` is brought into scope on one process, it is not on any other:

```julia
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
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
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
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. `@spawn` (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
julia> A = rand(1000,1000);

julia> Bref = @spawn A^2;

[...]

julia> fetch(Bref);
```

Method 2:

```julia
julia> Bref = @spawn rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawn`. 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 `@spawn` 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 `@spawn` statement just for this step.

25.4   Global variables

Expressions executed remotely via `@spawn`, 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 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 @spawn 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 = @spawn count_heads(100000000)
Future(2, 1, 6, nothing)

julia> b = @spawn 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 @spawn 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:

```julia
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:

```julia
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:

```julia
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
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
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)
```

```julia
            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,2)) seconds on worker $where")
            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 Futures are write-once and cached locally, the act of fetching 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 Futures, 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   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).

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
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
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
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
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
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
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 `SharedArrays` and time these functions, we get the following results (with `julia -p 4`):

```julia
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.

25.9   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:

```julia
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:

```julia
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=`-e <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.

- `ClusterManagers` 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.

25.10 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.

25.11   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 done to the 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
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.

```julia
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` repeteavely 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
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 http://mpi-forum.org/docs.

[2]Julia GPU man pages

Chapter 26

Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```julia
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
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
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
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
julia> collect(`echo "foo bar"`)
2-element Array{String,1}:
 "echo"
 "foo bar"

julia> `echo "foo bar"`[2]
"foo bar"
```

26.1   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
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
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
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`
```

26.2   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.

26.3   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 27

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. (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. When both libraries and executables are generated by LLVM, it is possible to perform whole-program optimizations that can even optimize across this boundary, but Julia does not yet support that. In the future, however, it may do so, yielding even greater performance gains.)

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. `library` refers to the shared library name: shared libraries available in the (platform-specific) load path will be resolved by name, and if necessary a direct path may 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. Arguments to `ccall` are as follows:

1. A (`:function`, `"library"`) pair, which must be written as a literal constant,

   OR

   a function pointer (for example, from `dlsym`).

2. Return type (see below for mapping the declared C type to Julia)

   – This argument will be evaluated at compile-time, when the containing method is defined.

3. A tuple of input types. The input types must be written as a literal tuple, not a tuple-valued variable or expression.

   – This argument will be evaluated at compile-time, when the containing method is defined.

4. The following arguments, if any, are the actual argument values passed to the function.

As a complete but simple example, the following calls the `clock` function from the standard C library:

```julia
julia> t = ccall((:clock, "libc"), 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 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
julia> path = ccall((:getenv, "libc"), 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
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`:

```julia
function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"),
                Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    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
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:

```julia
function gethostname()
    hostname = Vector{UInt8}(128)
    ccall((:gethostname, "libc"), Int32,
          (Ptr{UInt8}, Csize_t),
          hostname, sizeof(hostname))
    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.

27.1 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. Arguments to `@cfunction` are as follows:

1. A Julia Function

2. Return type

3. A literal tuple of input types

Like ccall, all of these arguments will be evaluated at compile-time, when the containing method is defined.

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 that works for some arbitrary objects (which define <):

```julia
julia> function mycompare(a, b)::Cint
           return (a < b) ? -1 : ((a > b) ? +1 : 0)
       end
mycompare (generic function with 1 method)
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a C int, so we annotate the return type of the function to be sure it returns a `Cint`.

In order to pass this function to C, we obtain its address using the macro `@cfunction`:

```julia
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
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 knows how to convert an array into a `Ptr{Cdouble}`, how to compute the size of a type in bytes (identical to C's `sizeof` operator), 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).

27.2   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. (The Clang package can be used to auto-generate Julia code from a C header file.)

Auto-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, a review of some relevant Julia type terminology:

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.

Syntax / Keyword	Example	Description
`mutable struct`	`String`	"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.

- `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:

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.

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)

System-dependent:

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

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)

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

```julia
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 a near 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:

```
in C:
struct B {
    int A[3];
};
b_a_2 = B.A[2];

in Julia:
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 `libc` library and cause Julia to crash. 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.

27.3 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:

```julia
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.

Special Reference Syntax for ccall (deprecated):

The & syntax is deprecated, use the `Ref{T}` argument type instead.

A prefix & is used on an argument to `ccall` to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself (required for all Fortran function arguments, as noted above). The following example computes a dot product using a BLAS function.

```julia
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
```

The meaning of prefix & is not quite the same as in C. In particular, any changes to the referenced variables will not be visible in Julia unless the type is mutable (declared via `mutable struct`). However, even for immutable structs it will not cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Moreover, & may be used with any expression, such as &0 or &f(x).

When a scalar value is passed with & as an argument of type Ptr{T}, the value will first be converted to type T.

27.4   Some Examples of C Wrappers

Here is a simple example of a C wrapper that returns a `Ptr` type:

```julia
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 (and not Julia).

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 should 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:

```julia
# 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 can be used in the same (covariant) context of the input argument to a `ccall`. A pointer to memory allocated by Julia must be of type `Ref{gsl_permutation}`, to ensure that the memory address pointed to is valid and that Julia's garbage collector manages the chunk of memory pointed to correctly. Therefore, the `Ref{gsl_permutation}` declaration allows pointers managed by C or Julia to be used.

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:

```julia
# 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}(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 can only be used with corresponding input type declaration `Ref{Cdouble}`, since its memory is allocated and managed by Julia, not C. 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.

Note that for this code to work correctly, `result_array` must be declared to be of type `Ref{Cdouble}` and not `Ptr{Cdouble}`. The memory is managed by Julia and the `Ref` signature alerts Julia's garbage collector to keep managing the memory for `result_array` while the `ccall` executes. If `Ptr{Cdouble}` were used instead, the `ccall` may still work, but Julia's garbage collector would not be aware that the memory declared for `result_array` is being used by the external C function. As a result, the code may produce a memory leak if `result_array` never gets freed by the garbage collector, or if the garbage collector prematurely frees `result_array`, the C function may end up throwing an invalid memory access exception.

27.5 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.

27.6   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.

27.7   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, ())
```

27.8   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.

```julia
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
```

27.9   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:

```julia
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.

27.10   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:

```julia
hn = Vector{UInt8}(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:

```julia
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`.

27.11   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
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!`.

27.12   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.

27.13 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.

27.14 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

27.15 C++

For direct C++ interfacing, see the Cxx package. For tools to create C++ bindings, see the CxxWrap package.

Chapter 28

Handling Operating System Variation

When dealing with platform libraries, it is often necessary to provide special cases for various platforms. The variable `Sys.KERNEL` can be used to write these special cases. There are several functions in the `Sys` module intended to make this easier: `isunix`, `islinux`, `isapple`, `isbsd`, and `iswindows`. These may be used as follows:

```
if Sys.iswindows()
    some_complicated_thing(a)
end
```

Note that `islinux` and `isapple` are mutually exclusive subsets of `isunix`. Additionally, there is a macro `@static` which makes it possible to use these functions to conditionally hide invalid code, as demonstrated in the following examples.

Simple blocks:

```
ccall((@static Sys.iswindows() ? :_fopen : :fopen), ...)
```

Complex blocks:

```
@static if Sys.islinux()
    some_complicated_thing(a)
else
    some_different_thing(a)
end
```

When chaining conditionals (including `if/elseif/end`), the `@static` must be repeated for each level (parentheses optional, but recommended for readability):

```
@static Sys.iswindows() ? :a : (@static Sys.isapple() ? :b : :c)
```

Chapter 29

Environment Variables

Julia may be configured with a number of environment variables, either in the usual way of the operating system, or in a portable way from within Julia. Suppose you want to set the environment variable JULIA_EDITOR to vim, then either 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 is determined by evaluating ENV["JULIA_EDITOR"].

The environment variables that Julia uses generally start with JULIA. If InteractiveUtils.versioninfo is called with verbose equal to true, then the output will list defined environment variables relevant for Julia, including those for which JULIA appears in the name.

29.1   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_LOAD_PATH

A separated list of absolute paths that are to be appended to the variable LOAD_PATH. (In Unix-like systems, the path separator is :; in Windows systems, the path separator is ;.) The LOAD_PATH variable is where Base.require and Base.load_in_path() look for code; it defaults to the absolute path $JULIA_HOME/../share/julia/stdlib/v$(VERSION.major).$(VERS so that, e.g., version 0.7 of Julia on a Linux system with a Julia executable at /bin/julia will have a default LOAD_PATH of /share/julia/stdlib/v0.7.

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

```
$HOME/.julia/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.

29.2   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.

29.3 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 `Base.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.

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.

29.4 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.

29.5   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 nth 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 nth 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 30

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#).

30.1   High-Level Embedding

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`.

30.2 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.

30.3 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.

30.4   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 is working on the stack, so it must be exactly paired with a JL_GC_POP before the stack frame is destroyed.

Several Julia values can be pushed at once using the JL_GC_PUSH2 , JL_GC_PUSH3 , and JL_GC_PUSH4 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();
```

The garbage collector also 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

30.5   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.

30.6   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 31

Code Loading

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. It should be noted, however, that `import X` can load different packages in different contexts: X can refer to one package named X in the main project but potentially different packages named X in each dependency. More on this below.

Code inclusion is quite straightforward: it simply parses and evaluates a source file in the context of the caller. Package loading is built on top of code inclusion and is quite a bit more complex. The rest of this chapter, therefore, focuses on the behavior and mechanics of package loading.

> **Note**
>
> You only need to read this chapter if you want to understand the technical details of package loading in Julia. If you just want to install and use packages, simply use Julia's built-in package manager to add packages to your environment and write `import X` or `using X` in your code to load packages that you've added.

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. The effect of `import X` depends on two questions:

1. **What** package is X in this context?

2. **Where** can that X package be found?

Understanding how Julia answers these questions is key to understanding package loading.

31.1   Federation of packages

Julia supports federated management of packages. This means that multiple independent parties can maintain both public and private packages and registries of them, 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 ships with Julia 0.7/1.0 and lets you install and manage dependencies of your projects, by creating and manipulating project files, which describe what your project depends on, and manifest files that 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 can quite possibly end up depending on different packages with the same name. Julia's package loading mechanism handles this by not requiring 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 are assigned to them before they are registered. The question *"what is X?"* is answered by determining the UUID of X.

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 that name. 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—through no action of yours other than upgrading—depending on two different packages named `Priv`. 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 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. Julia's package loading mechanism allows this by distinguishing the two `Priv` packages by context and UUID. How this distinction works is determined by environments, as explained in the following sections.

31.2   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 three kinds of environments:

1. **A project environment** is a directory with a project file and an optional manifest file. 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. This kind of environment was the only kind that existed in Julia 0.6 and earlier. 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.

3. **A stacked environment** is an ordered set of project environments and package directories, overlaid to make a single composite environment in which all the packages available in its constituent environments are available. Julia's load path is a stacked environment, for example.

These three kinds of 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 since the manifest file captures the exact version of every dependency and can be rematerialized easily.

- Package directories provide low-overhead **convenience** when a project environment would be overkill: are handy when you have a set of packages and just want to put them somewhere and use them as they are without having to create and maintain a project environment for them.

- Stacked environments allow for **augmentation** of the primary environment with additional tools. You can push an environment including development tools onto the stack and they will be available from the REPL and scripts but not from inside of packages.

As an abstraction, an environment provides three maps: `roots`, `graph` and `paths`. When resolving the meaning of `import X`, `roots` and `graph` are used to determine the identity of X and answer the question *"what is X?"*, while the `paths` map is used to locate the source code of X and answer the question *"where is 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 the entry-point source file of that package. 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 an dependency), Julia determines what file to load to acquire X by looking up `paths[uuid, :X]` in the environment. Including this file should create a module named X. After the first time this package is loaded, any import resolving to the same `uuid` will simply create a new binding to the same already-loaded package module.

Each kind of environment defines these three maps differently, as detailed in the following sections.

> **Note**
>
> For clarity of exposition, the examples throughout this chapter include fully materialized data structures for `roots`, `graph` and `paths`. However, these maps are really only abstractions—for efficiency, Julia's package loading code does not actually materialize them. Instead, it queries them through internal APIs and lazily computes only as much of each structure as is necessary to load a given package.

Project environments

A project environment is determined by a directory containing a project file, `Project.toml`, and optionally a manifest file, `Manifest.toml`. These files can also be named `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 named `Project.toml` and `Manifest.toml` significant. For pure Julia projects, however, the names `Project.toml` and `Manifest.toml` should be 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 above:

```
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 were materialized as 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 the code of App 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, or if there is no manifest file, `graph` is empty. A manifest file contains a stanza for each direct or indirect dependency of a project, including for each one, its 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 `Priv` packages that the application needs—a private one which is a direct dependency and a public one which is an indirect dependency through Pub:

    - 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 which the private `Priv` package depends on.

A materialized representation of this dependency `graph` looks like this:

```julia
graph = Dict{UUID,Dict{Symbol,UUID}}(
    # Priv – the private one:
    UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b") => Dict{Symbol,UUID}(
        :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{Symbol,UUID}(),
    # Pub:
    UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1") => Dict{Symbol,UUID}(
        :Priv  => UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Zebra:
    UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62") => Dict{Symbol,UUID}(),
)
```

Given this dependency `graph`, when Julia sees `import Priv` in the Pub package—which has UUID c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1—it looks up:

```julia
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 the packages dependencies, which allows for name collisions 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 also determined by the manifest file if present and is empty if there is no manifest. The path of a package `uuid` named X is determined by these two rules:

1. If the manifest stanza matching `uuid` has a `path` entry, use that path relative to the manifest file.

2. Otherwise, if the manifest stanza matching `uuid` has a `git-tree-sha1` entry, compute a deterministic hash function of `uuid` and `git-tree-sha1`—call it `slug`—and look for `packages/X/$slug` in each directory in the Julia `DEPOT_PATH` global array. Use the first such directory that exists.

If applying these rules doesn't find a loadable path, the package should be considered not installed and the system should raise an error or prompt the user to install the appropriate package version.

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 `HDkr` (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/HDkr/src/Priv.jl` in one of the package depots. Suppose the contents of `DEPOT_PATH` is `["/users/me/.julia", "/usr/local/julia"]`; then Julia will look at the following paths to see if they exist:

1. `/home/me/.julia/packages/Priv/HDkr/src/Priv.jl`

2. `/usr/local/julia/packages/Priv/HDkr/src/Priv.jl`

Julia uses the first of these that exists to load the public `Priv` package.

Here is a materialized `paths` map for the App project environment:

```julia
paths = Dict{Tuple{UUID,Symbol},String}(
    # 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:

1. The private `Priv` package is "vendored" inside of 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 kind of environment that approximates package loading in Julia 0.6 and earlier, and which resembles package loading in many other dynamic languages. The set of packages available in a package directory corresponds to the set of subdirectories it contains that look like packages: if `X/src/X.jl` is a file in a package directory, then X is considered to be a package and `X/src/X.jl` is the file you load to get X. Which packages can "see" each other as dependencies depends on whether they contain project files or not and what appears in the `[deps]` sections of those project files.

The roots map is determined by the subdirectories X of a package directory for which `X/src/X.jl` exists and whether `X/Project.toml` exists and has a top-level uuid entry. Specifically `:X => uuid` goes in `roots` for each such X where uuid is defined as:

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 path of `X/Project.toml`.

3. If `X/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 subdirectory 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, materialized as a dictionary:

```
roots = Dict{Symbol,UUID}(
    :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, materialized as a dictionary:

```
graph = Dict{UUID,Dict{Symbol,UUID}}(
    # Bobcat:
    UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf") => Dict{Symbol,UUID}(
        :Cobra => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"),
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Cobra:
    UUID("4725e24d-f727-424b-bca0-c4307a3456fa") => Dict{Symbol,UUID}(
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Dingo:
    UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc") => Dict{Symbol,UUID}(),
)
```

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 possibly 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 would be materialized as this dictionary:

```
paths = Dict{Tuple{UUID,Symbol},String}(
    (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 pushing an environment containing these tools onto 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 simply merged as dictionaries, favoring earlier entries over later ones in the case of key collisions. In other words, if we have stack = [env, env, …] then we have:

```
roots = reduce(merge, reverse([roots, roots, …]))
graph = reduce(merge, reverse([graph, graph, …]))
paths = reduce(merge, reverse([paths, paths, …]))
```

The subscripted roots, graph and paths variables correspond to the subscripted environments, env, contained stack. The reverse is present because merge favors the last argument rather than first when there are collisions between keys in its argument dictionaries. That's all there is to stacked environments. 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.

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 tradeoff: it's better to break your dev 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.

31.3   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. Fortunately, most Julia users can remain oblivious to the technical details of code loading and simply use the built-in package manager to add a package X to the appropriate project and manifest files and then write `import  X` to load X without a further thought.

Chapter 32

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 (in contrast to the alternative instrumenting profiler).

- 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.

32.1 Basic usage

Let's work with a simple test case:

```julia
julia> function myfunc()
           A = rand(200, 200, 400)
           maximum(A)
       end
```

345

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.

32.2   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().

32.3   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`:

```julia
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

32.4 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:

```julia
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 33

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 34

Stack Traces

The StackTraces module provides simple stack traces that are both human readable and easy to use programmatically.

34.1   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
```

34.2   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.

34.3   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
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
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
julia> @noinline child() = error("Whoops!")
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)
```

```julia
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
[...]
```

34.4   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
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
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 35

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.

35.1   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 toplevel will be a **global** variable. Variables defined in at top level scope inside modules are also global.

In the following REPL session:

```
julia> x = 1.0
```

is equivalent to:

```julia
julia> global x = 1.0
```

so all the performance issues discussed previously apply.

35.2   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
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
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.

35.3 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.

35.4 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 = 3.1415926535897...
```

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.

35.5   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
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
julia> mutable struct MyType{T<:AbstractFloat}
           a::T
       end
```

This is a better choice than

```julia
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
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
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
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
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
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.

Declare types of keyword arguments

Keyword arguments can have declared types:

```
function with_keyword(x; name::Int = 1)
    ...
end
```

Functions are specialized on the types of keyword arguments, so these declarations will not affect performance of code inside the function. However, they will reduce the overhead of calls to the function that include keyword arguments.

Functions with keyword arguments have near-zero overhead for call sites that pass only positional arguments.

Passing dynamic lists of keyword arguments, as in f(x; keywords...), can be slow and should be avoided in performance-sensitive code.

35.6   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.

35.7   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.

35.8   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`

35.9   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.

35.10   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
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
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:

```julia
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.

35.11 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 crazy 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},

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.

35.12 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
```

```julia
        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
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, Any[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.

35.13   Pre-allocating outputs

If your function returns an `Array` or some other complex type, it may have to allocate memory. Unfortunately, often-times 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
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
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).

35.14   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.)

35.15 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
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.

35.16 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
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.

35.17   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))
```

35.18   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(pid, foo, 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.

35.19   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.

35.20   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)`.

35.21   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):

```julia
@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:

```julia
function init!(u::Vector)
    n = length(u)
    dx = 1.0 / (n-1)
```

```julia
    @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/n)
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 NaNs will not occur during the computation, which can lead to surprising behavior:

```julia
julia> f(x) = isnan(x);

julia> f(NaN)
true

julia> f_fast(x) = @fastmath isnan(x);

julia> f_fast(NaN)
false
```

35.22   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:

```julia
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
```

35.23 @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)
           sin(y*x + 1)
       end;

julia> @code_warntype f(3.2)
Body::Float64
2 1  %1  = invoke Main.pos(%%x::Float64)::UNION{FLOAT64, INT64}
3    %2  = isa(%1, Float64)::Bool
         goto 3 if not %2
```

```
2   %4  = π (%1, Float64)
    %5  = Base.mul_float(%4, %%x)::Float64
          goto 6
3   %7  = isa(%1, Int64)::Bool
          goto 5 if not %7
4   %9  = π (%1, Int64)
    %10 = Base.sitofp(Float64, %9)::Float64
    %11 = Base.mul_float(%10, %%x)::Float64
          goto 6
5         Base.error("fatal error in type inference (type bound)")
          unreachable
6   %15 = φ (2 => %5, 4 => %11)::Float64
    %16 = Base.add_float(%15, 1.0)::Float64
    %17 = invoke Main.sin(%16::Float64)::Float64
          return %17
```

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*xis 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, Unions 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`

35.24   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 36

Workflow Tips

Here are some tips for working with Julia efficiently.

36.1 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

  <your definitions here>

  end
  ```

- **Put your test code in another file.** Create another file, say `tst.jl`, which begins with

  ```
  import Tmp
  ```

  and includes tests for the contents of Tmp. Alternatively, you can wrap the contents of your test file in a module, as

  ```
  module Tst
      using Tmp

      <scratch work>

  end
  ```

  The advantage is that you can now do `using  Tmp` in your test code and can therefore avoid prepending `Tmp.` everywhere. The disadvantage is that code can no longer be selectively copied to the REPL without some tweaking.

- **Lather. Rinse. Repeat.** Explore ideas at the `julia` command prompt. Save good ideas in `tst.jl`.

Simplify initialization

To simplify restarting the REPL, put project-specific initialization code in a file, say `_init.jl`, which you can run on startup by issuing the command:

```
julia -L _init.jl
```

If you further add the following to your `~/.julia/config/startup.jl` file

```
isfile("_init.jl") && include(joinpath(pwd(), "_init.jl"))
```

then calling `julia` from that directory will run the initialization code without the additional command line argument.

36.2   Browser-based workflow

It is also possible to interact with a Julia REPL in the browser via IJulia. See the package home for details.

Chapter 37

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.

37.1 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`).

37.2 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.

37.3   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.

37.4   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.

37.5   Avoid strange type `Union`s

Types such as `Union{Function,AbstractString}` are often a sign that some design could be cleaner.

37.6   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.

37.7   Use naming conventions consistent with Julia's `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.

37.8   Write functions with argument ordering similar to Julia's 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 `Vararg`s, 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.

37.9   Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

37.10   Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

37.11   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.

37.12   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.

37.13 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.

37.14 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.

37.15 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.

37.16 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.

37.17   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 Symbols 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.

37.18   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.

37.19   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).

37.20   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 38

Frequently Asked Questions

38.1 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)
...
```

38.2   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
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
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
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
```

38.3   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
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
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
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
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
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
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
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
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
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
```

38.4 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).

38.5 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.

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`.

38.6   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.

38.7   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()
ReentrantLock(nothing, Condition(Any[]), 0)

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
```

38.8   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.

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.

38.9   Julia Releases

Do I want to use a release, beta, or nightly version of Julia?

You may prefer the release version of Julia if you are looking for a stable code base. Releases generally occur every 6 months, giving you a stable platform for writing code.

You may prefer the beta version of Julia if you don't mind being slightly behind the latest bugfixes and changes, but find the slightly faster rate of changes more appealing. Additionally, these binaries are tested before they are published to ensure they are fully functional.

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.

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.

When are deprecated functions removed?

Deprecated functions are removed after the subsequent release. For example, functions marked as deprecated in the 0.1 release will not be available starting with the 0.2 release.

Chapter 39

Noteworthy Differences from other Languages

39.1 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 Vectors 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. Arbitrarily large integer literals are supported. As a result, some operations such as 2^-1 will throw 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. MAT-LAB'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 The return Keyword 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 used 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 <, > and =.

- 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 structs do not support dynamically adding fields at runtime, unlike MATLAB's classes. 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...).

39.2   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 one or two arguments. The first argument is an iterable collection such as an array. If there is a second argument, then this argument 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) instead of isna(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.

39.3 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.

39.4   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 now 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).

Chapter 40

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.

Part IV

Base

Chapter 41

Essentials

41.1 Introduction

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.

Some general notes:

- To use module functions, use `import Module` to import the module, and `Module.fn(x)` to use the functions.

- Alternatively, `using Module` will import all exported `Module` functions into the current namespace.

- By convention, function names ending with an exclamation point (`!`) modify their arguments. Some functions have both modifying (e.g., `sort!`) and non-modifying (`sort`) versions.

41.2 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.

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 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(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.

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([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 1-argument 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.

source

Base.MainInclude.include – *Function.*

```
include(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 1-argument 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.

Use Base.include to evaluate a file into another module.

source

Base.include_string – *Function.*

```
include_string(m::Module, code::AbstractString, filename::AbstractString="string")
```

Like `include`, except reads code from the given string rather than from a file.

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])
```

Returns the method table for `f`.

If `types` is specified, returns an array of methods whose types match.

source

Base.@show – Macro.

> @show

Show an expression and result, returning the result.

> source

ans – Keyword.

> ans

A variable referring to the last computed value, automatically set at the interactive prompt.

> source

41.3   Keywords

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 exported 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 exported 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 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. For example:

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!
```

source

`return` – Keyword.

```
return
```

`return` can be used in function bodies to exit early and return a given value, e.g.

```
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 its enclosing function 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]`.

source

`do` – Keyword.

```
do
```

Create an anonymous function. 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 the body of the loop by 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 continues evaluating the body of the while loop so 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 for `Exceptions` to be tested for. For example, a customized square root function can be written to automatically call either the real or complex square root method on demand using `Exceptions`:

```
f(x) = try
    sqrt(x)
catch
    sqrt(complex(x, 0))
end
```

try/catch statements also allow the `Exception` to be saved in a variable, e.g. `catch y`.

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 which are also constant. 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 as constants.

Note that "constant-ness" is not enforced inside containers, so if x is an array or dictionary (for example) you can still add and remove elements.

Technically, you can even redefine const variables, although this will generate a warning from the compiler. The only strict requirement is that the *type* of the variable does not change, which is why const variables are much faster than regular globals.

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

`...` – 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

41.4   Base Modules

Base – Keyword.

The base library of Julia.

source

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.

Experimental multithreading support.

source

Base.GC – Module.

```
Base.GC
```

Module with garbage collection utilities.

source

41.5   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 canonical total order. 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 canonical total order should implement this function. Numeric types only need to implement it if they have special values such as NaN. Types with a canonical partial order should implement `<`.

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.

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 = 3.1415926535897...)
```

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.

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.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

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).

source

Base.setproperty! – Function.

```
setproperty!(value, name::Symbol, x)
```

The syntax a.b = c calls setproperty!(a, :b, c).

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

Core.getfield – Function.

```
getfield(value, name::Symbol)
```

Extract a named field from a value of composite type. See also getproperty.

Examples

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1

julia> a.num
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: type Rational is immutable
```

source

Core.isdefined – Function.

```
isdefined(m::Module, s::Symbol)
isdefined(object, s::Symbol)
isdefined(object, index::Int)
```

Tests whether an assignable location is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index.

Examples

```julia
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.

```julia
@isdefined s -> Bool
```

Tests whether variable s is defined in the current scope.

Examples

```julia
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.

```julia
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(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

41.6 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
```

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.

```
Base.isabstracttype(T)
```

Determine whether type T was declared as an abstract type (i.e. using the abstract keyword).

Examples

```julia
julia> Base.isabstracttype(AbstractArray)
true

julia> Base.isabstracttype(Vector)
false
```

source

Base.isprimitivetype – Function.

```
Base.isprimitivetype(T) -> Bool
```

Determine whether type T was declared as a primitive type (i.e. using the primitive keyword).

source

Base.isstructtype – Function.

```
Base.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
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
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

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: argument is an abstract type; size is indeterminate
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.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)
```

The smallest in absolute value non-subnormal value representable by the given floating-point DataType T.

source

Base.floatmax – Function.

```
floatmax(T)
```

The highest finite value representable by the given floating-point DataType T.

Examples

```julia
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38
```

source

Base.maxintfloat – Function.

```julia
maxintfloat(T=Float64)
```

The largest consecutive integer that is exactly represented in the given floating-point type T (which defaults to Float64).

That is, maxintfloat returns the smallest positive integer n such that n+1 is *not* exactly representable in the type T.

source

```julia
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.

```julia
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

source

Base.eps – Method.

```julia
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

```julia
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

$$|y - x| \leq \mathrm{eps}(x)/2.$$

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::Color = 0, blue::Color = 1, green::Color = 2)
```

source

41.7   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
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
```

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
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
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
```

NamedTuples 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.

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
```

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 `nothing`.

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.

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 – *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"
```

Values can also be specified inside a begin block, e.g.

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.

source

41.8 Generic Functions

`Core.Function` – Type.

> Function

Abstract type of all functions.

> source

`Base.hasmethod` – Function.

> hasmethod(f, Tuple type; 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`.

Examples

> source

`Core.applicable` – Function.

> applicable(f, args...) -> Bool

Determine whether the given generic function has a method applicable to the given arguments.

Examples

> 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).

Examples

> 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

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>.

Examples

source

41.9   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 Base.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.

> **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

> **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.@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:

source

Base.@noinline – Macro.

```
@noinline
```

Prevents the compiler from inlining 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:

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.

source

Base.@specialize – Macro.

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see @specialize.

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`](@ref) 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`](@ref)
  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:

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.

source

41.10   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`.

Examples

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.

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

source

41.11 System

Base.run – Function.

```
run(command, args...; wait::Bool = true)
```

Run a command object, constructed with backticks. 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:

source

Base.success – Function.

```
success(command)
```

Run a command object, constructed with backticks, 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=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 file-names. 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:

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:

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

`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.

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.

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.

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.

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. Note: the expression is evaluated inside a local function, instead of the current context, in order to eliminate the effects of compilation, however, there still may be some allocations due to JIT compilation. This also makes the results inconsistent with the @time macros, which do not try to adjust for the effects of compilation.

See also @time, @timev, @timed, and @elapsed.

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.

> 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.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.@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]

41.12   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

[source]

`Base.@v_str` – Macro.

```
@v_str
```

String macro used to parse a string to a `VersionNumber`.

Examples

[source]

41.13   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([e])
```

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a catch block).

> 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.@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

> 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

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

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

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.

Example

source

`Core.InexactError` – Type.

> `InexactError(name::Symbol, T, val)`

Cannot exactly convert `val` to type `T` in a method of function name.

Examples

source

`Core.InterruptException` – Type.

> `InterruptException()`

The process was stopped by a terminal interrupt (CTRL+C).

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

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.

> source

`Core.UndefRefError` – Type.

> `UndefRefError()`

The item or field is not defined for the given object.

> source

`Core.UndefVarError` – Type.

> `UndefVarError(var::Symbol)`

A symbol in the current scope is not defined.

Examples

> 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::Function;  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`.

Examples

> 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

41.14 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.

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

41.15 Reflection

Base.nameof – Method.

```
nameof(m::Module) -> Symbol
```

Get the name of a `Module` as a `Symbol`.

Examples

source

Base.parentmodule – Function.

```
parentmodule(m::Module) -> Module
```

Get a module's enclosing `Module`. `Main` is its own parent.

Examples

source

```
parentmodule(t::DataType) -> Module
```

Determine the module containing the definition of a (potentially `UnionAll`-wrapped) `DataType`.

Examples

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 `m.jl` file that was used to `import` module `m`, or `nothing` if `m` was not imported from a package.

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

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

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, or `:anonymous`.

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

41.16   Internals

`Base.GC.gc` – Function.

```
GC.gc()
```

Perform garbage collection.

> **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
```

Temporarily protect the given objects from being garbage collected, even if they would otherwise be unreferenced.

The last argument is the expression during which the object(s) will be preserved. The previous arguments are the objects to preserve.

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.

[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.

[source](#)

Base.Meta.ParseError – Type.

```
ParseError(msg)
```

The expression passed to the parse function could not be interpreted as a valid Julia expression.

[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:

[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:

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)

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.

Note that an error will be thrown if types are not leaf types when generated is true and the corresponding method is a @generated method.

> source

Base.code_typed – Function.

> code_typed(f, types; optimize=true)

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.

> 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

Chapter 42

Collections and Data Structures

42.1 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.

> 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.

> source

Fully implemented by:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `BitSet`
- `IdDict`
- `Dict`
- `WeakKeyDict`
- `EachLine`
- `AbstractString`
- `Set`
- `Pair`
- `NamedTuple`

42.2 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

> 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 `Integers` creates a `UnitRange`.

Examples

> 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

> source

42.3   General Collections

Base.isempty – Function.

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

Examples

source

Base.empty! – Function.

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

Examples

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

source

Fully implemented by:

- AbstractRange

- UnitRange

- Tuple

- Number

- AbstractArray

- BitSet

- IdDict

- Dict

- WeakKeyDict

- AbstractString

- Set

- NamedTuple

42.4 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, `Set`s check whether the item `isequal` to one of the elements. `Dict`s 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`.

Examples

source

Base.: – Function.

```
(item, collection) -> Bool
(collection, item) -> Bool
```

Negation of and , i.e. checks that `item` is not in `collection`.

Examples

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

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

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

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

source

```
unique(A::AbstractArray; dims::Int)
```

Return unique regions of `A` along dimension `dims`.

Examples

source

Base.unique! – Function.

```
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

source

Base.allunique – Function.

```
allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

Examples

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

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

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

source

`Base.maximum` – Function.

```
maximum(itr)
```

Returns the largest element in a collection.

Examples

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

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

source

`Base.minimum` – Function.

```
minimum(itr)
```

Returns the smallest element in a collection.

Examples

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

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

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

source

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

Examples

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

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

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

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

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

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

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

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

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

Note the important difference between sum(A) and reduce(+, A) for arrays with small integer eltype:

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

source

```
sum(A::AbstractArray; dims)
```

Sum elements of an array over the given dimensions.

Examples

source

Base.sum! – Function.

```
sum!(r, A)
```

Sum elements of A over the singleton dimensions of r, and write results to r.

Examples

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

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

source

```
prod(A::AbstractArray; dims)
```

Multiply elements of an array over the given dimensions.

Examples

source

`Base.prod!` – Function.

```
prod!(r, A)
```

Multiply elements of A over the singleton dimensions of `r`, and write results to `r`.

Examples

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

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

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

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

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

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

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

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

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

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

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

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

source

Base.mapreduce – Method.

```
mapreduce(f, op, itr; [init])
```

Apply function f to each element in `itr`, and then reduce the result using the binary function op. If provided, `init` must be a neutral element for op that will be returne 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.

Examples

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

source

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

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

source

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last n characters of s.

source

`Base.step` – Function.

```
step(r)
```

Get the step size of an `AbstractRange` object.

Examples

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

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

source

`Base.filter` – Function.

```
filter(f, a::AbstractArray)
```

Return a copy of a, removing elements for which f is `false`. The function f is passed one argument.

Examples

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

source

Base.filter! – Function.

```
filter!(f, a::AbstractVector)
```

Update a, removing elements for which f is false. The function f is passed one argument.

Examples

source

```
filter!(f, d::AbstractDict)
```

Update d, removing elements for which f is false. The function f is passed key=>value pairs.

Example

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

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

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

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

source

42.5   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

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

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

source

Fully implemented by:

- `Array`

- `BitArray`

- `AbstractArray`

- `SubArray`

Partially implemented by:

- `AbstractRange`

- UnitRange

- Tuple

- AbstractString

- Dict

- IdDict

- WeakKeyDict

- NamedTuple

42.6 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.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

Alternatively, a sequence of pair arguments may be passed.

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, and thus 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 Immutable Dictionary for the 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

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

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

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

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

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:

source

`Base.getkey` – Function.

```
getkey(collection, key, default)
```

Return the key matching argument key if one exists in `collection`, otherwise return `default`.

Examples

source

`Base.delete!` – Function.

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, and return the collection.

Examples

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

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

source

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. Since the values are stored internally in a hash table, 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

source

Base.pairs – Function.

```
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

```
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

See also: IndexStyle, axes.

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

source

```
merge(combine, 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. Values with the same key will be combined using the combiner function.

Examples

source

```
merge(a::NamedTuple, b::NamedTuple)
```

Construct a new named tuple by merging two existing ones. The order of fields in a is preserved, but values are taken from matching fields in b. Fields present only in b are appended at the end.

source

```
merge(a::NamedTuple, iterable)
```

Interpret an iterable of key-value pairs as a named tuple, and perform a merge.

source

Base.merge! – Method.

```
merge!(d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. See also merge.

Examples

source

Base.merge! – Method.

```
merge!(combine, d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

Examples

source

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(type)
```

Get the key type of an dictionary type. Behaves similarly to `eltype`.

Examples

source

`Base.valtype` – Function.

```
valtype(type)
```

Get the value type of an dictionary type. Behaves similarly to `eltype`.

Examples

source

Fully implemented by:

- `IdDict`

- `Dict`

- `WeakKeyDict`

Partially implemented by:

- `BitSet`

- `Set`

- `EnvDict`

- `Array`

- `BitArray`

- `ImmutableDict`

- `Iterators.Pairs`

42.7   Set-Like Collections

`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 `Ints` 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

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

source

Base.intersect – Function.

```
intersect(s, itrs...)
(s, itrs...)
```

Construct the intersection of sets. Maintain order with arrays.

Examples

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

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

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

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)
(a,b)  -> Bool
(b, a) -> Bool
```

Determine whether every element of a is also in b, using in.

Examples

source

`Base.:` – Function.

```
(a, b)
(b, a)
```

Negation of and , i.e. checks that a is not a subset of b.

Examples

source

`Base.:` – Function.

```
(a, b)
(b, a)
```

Determines if a is a subset of, but not equal to, b.

Examples

source

`Base.issetequal` – Function.

```
issetequal(a, b)
```

Determine whether a and b have the same elements. Equivalent to a ⊆ b && b ⊆ a.

Examples

source

Fully implemented by:

- BitSet

- Set

Partially implemented by:

- Array

42.8   Dequeues

Base.push! – Function.

```
push!(collection, items...) -> collection
```

Insert one or more `items` at the end of `collection`.

Examples

Use append! to add all the elements of another collection to `collection`. The result of the preceding example is equivalent to append!([1, 2, 3], [4, 5, 6]).

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

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

source

Base.pushfirst! – Function.

```
pushfirst!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

Examples

source

Base.popfirst! – Function.

```
popfirst!(collection) -> item
```

Remove the first `item` from `collection`.

Examples

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

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

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

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

To insert `replacement` before an index n without removing any items, use `splice!(collection, n:n-1, replacement)`.

source

```
splice!(a::Vector, range, [replacement]) -> items
```

Remove items in the specified index range, and return a collection containing the removed items. 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 items.

To insert `replacement` before an index n without removing any items, use `splice!(collection, n:n-1, replacement)`.

Examples

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

source

Base.append! – Function.

```
append!(collection, collection2) -> collection.
```

Add the elements of `collection2` to the end of `collection`.

Examples

Use push! to add individual items to `collection` which are not already themselves in another collection. The result is 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

source

Fully implemented by:

- `Vector` (a.k.a. 1-dimensional Array)

- `BitVector` (a.k.a. 1-dimensional BitArray)

42.9   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.

See also: Dict

Examples

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 43

Mathematics

43.1 Mathematical Operators

`Base.:-` – Method.

```
-(x)
```

Unary minus operator.

Examples

source

`Base.:+` – Function.

```
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.

source

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

Examples

source

`Base.:-` – Method.

```
-(x, y)
```

Subtraction operator.

Examples

source

`Base.:*` – Method.

```
*(x, y...)
```

Multiplication operator. x*y*z*... calls this function with all arguments, i.e. *(x, y, z, ...).

Examples

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

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

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.)

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

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

source

Base.div – Function.

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean division. Computes x/y, truncated to an integer.

Examples

source

Base.fld – Function.

```
fld(x, y)
```

Largest integer less than or equal to x/y.

Examples

source

Base.cld – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to x/y.

Examples

source

Base.mod – Function.

```
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.

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.

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

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

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.
- if r == RoundToZero, then the result is in the interval $[0, 2]$ if x is positive,. or $[-2, 0]$ otherwise.
- if r == RoundDown, then the result is in the interval $[0, 2]$.
- if r == RoundUp, then the result is in the interval $[-2, 0]$.

Examples

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.

Examples

source

`Base.divrem` – Function.

```
divrem(x, y)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x,y), rem(x,y))` or `(x÷y, x%y)`.

Examples

source

`Base.fldmod` – Function.

```
fldmod(x, y)
```

The floored quotient and modulus after division. 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

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

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

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

source

Base.`numerator` – Function.

```
numerator(x)
```

Numerator of the rational representation of x.

Examples

source

Base.`denominator` – Function.

```
denominator(x)
```

Denominator of the rational representation of x.

Examples

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

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

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 0s if `x >= 0`, 1s 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

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

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 0s. 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

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.

```
(:)(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.

source

Base.range – Function.

```
range(start; 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 (a LinRange).

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 (a StepRange).

Examples

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 reference value, a step, and the length. 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

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

source

`Base.:!==` – Function.

```
!==(x, y)
(x,y)
```

Always gives the opposite answer as `===`.

Examples

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

source

`Base.:<=` – Function.

```
<=(x, y)≤
(x,y)
```

Less-than-or-equals comparison operator. Falls back to `(x < y) | (x == y)`.

Examples

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

source

Base.:>= – Function.

```
>=(x, y)≥
(x,y)
```

Greater-than-or-equals comparison operator. Falls back to y <= x.

Examples

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

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

source

Base.:~ – Function.

```
~(x)
```

Bitwise not.

Examples

source

Base.:& – Function.

```
&(x, y)
```

Bitwise and. Implements three-valued logic, returning missing if one operand is missing and the other is true.

Examples

source

Base.:| – Function.

```
|(x, y)
```

Bitwise or. Implements three-valued logic, returning `missing` if one operand is `missing` and the other is `false`.

Examples

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

source

`Base.:!` – Function.

```
!(x)
```

Boolean not. Implements three-valued logic, returning `missing` if x is `missing`.

Examples

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

source

`&&` – Keyword.

```
x && y
```

Short-circuiting boolean AND.

source

`||` – Keyword.

```
x || y
```

Short-circuiting boolean OR.

source

43.2   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 `vecnorm` 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

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 $\cos(\pi x)$ 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 $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

source

Base.Math.cosc – Function.

```
cosc(x)
```

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ 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

source

Base.Math.rad2deg – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

Examples

source

Base.Math.hypot – Function.

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{x^2 + y^2}$ avoiding overflow and underflow.

Examples

source

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum x_i^2}$ avoiding overflow and underflow.

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.

source

Base.log – Method.

```
log(b,x)
```

Compute the base b logarithm of x. Throws DomainError for negative Real arguments.

Examples

> **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,

source

Base.log2 – Function.

```
log2(x)
```

Compute the logarithm of x to base 2. Throws `DomainError` for negative `Real` arguments.

Examples

source

`Base.log10` – Function.

```
log10(x)
```

Compute the logarithm of x to base 10. Throws `DomainError` for negative `Real` arguments.

Examples

source

`Base.log1p` – Function.

```
log1p(x)
```

Accurate natural logarithm of 1+x. Throws `DomainError` for `Real` arguments less than -1.

Examples

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

source

`Base.exp2` – Function.

```
exp2(x)
```

Compute the base 2 exponential of x, in other words 2^x.

Examples

source

`Base.exp10` – Function.

```
exp10(x)
```

Compute the base 10 exponential of x, in other words 10^x.

Examples

source

```
exp10(x)
```

Compute 10^x.

Examples

source

Base.Math.ldexp – Function.

```
ldexp(x, n)
```

Compute $x \times 2^n$.

Examples

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

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

> **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 44

Examples

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

> 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 RoundingModes. The first RoundingMode is used for rounding the real components while the second is used for rounding the imaginary components.

Example

> 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

[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

source

Base.minmax – Function.

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`. See also: extrema that returns `(minimum(x), maximum(x))`.

Examples

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

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

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

source

`Base.copysign` – Function.

```
copysign(x, y) -> z
```

Return z which has the magnitude of x and the same sign as y.

Examples

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

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

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

source

Base.isqrt – Function.

```
isqrt(n::Integer)
```

Integer square root: the largest integer m such that m*m <= n.

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

source

Base.real – Method.

> real(z)

Return the real part of the complex number z.

Examples

source

Base.imag – Function.

> imag(z)

Return the imaginary part of the complex number z.

Examples

source

Base.reim – Function.

> reim(z)

Return both the real and imaginary parts of the complex number z.

Examples

source

Base.conj – Function.

> conj(z)

Compute the complex conjugate of a complex number z.

Examples

source

Base.angle – Function.

> angle(z)

Compute the phase angle in radians of a complex number z.

Examples

source

Base.cis – Function.

> cis(z)

Return $\exp(iz)$.

Examples

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:

$$\binom{n}{k} =$$

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

See also

- `factorial`

External links

- Binomial coeffient 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

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).

Examples

source

Base.lcm – Function.

> lcm(x,y)

Least common (non-negative) multiple.

Examples

source

Base.gcdx – Function.

> gcdx(x,y)

Computes the greatest common (positive) divisor of x and y and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v).

Examples

> **Note**
>
> Bézout coefficients are *not* uniquely defined. gcdx returns the minimal Bézout 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

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

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

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

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

source

Base.powermod – Function.

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod{m}$.

Examples

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

source

Base.widemul – Function.

```
widemul(x, y)
```

Multiply x and y, giving the result as a larger type.

Examples

source

Base.Math.@evalpoly – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k c[k]z^{k-1}$ 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

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

source

Chapter 45

Numbers

45.1 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.

source

Concrete number types

Core.Float16 – Type.

> Float16 <: AbstractFloat

16-bit floating point number type.

source

Core.Float32 – Type.

> Float32 <: AbstractFloat

32-bit floating point number type.

source

Core.Float64 – Type.

> Float64 <: AbstractFloat

64-bit floating point number type.

source

Base.MPFR.BigFloat – Type.

> BigFloat <: AbstractFloat

Arbitrary precision floating point number type.

source

Core.Bool – Type.

> Bool <: Integer

Boolean type.

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.

source

Base.Irrational – Type.

```
Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol sym.

source

45.2   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

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

source

Base.bitstring – Function.

```
bitstring(n)
```

A string giving the literal bit representation of a number.

Examples

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.

Examples

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(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(x) -> Unsigned
```

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

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

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

source

`Base.bswap` – Function.

```
bswap(n)
```

Reverse the byte order of n.

Examples

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

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

source

45.3 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

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

source

`Base.zero` – Function.

```
zero(x)
```

Get the additive identity element for the type of x (x can also specify the type itself).

Examples

source

`Base.im` – Constant.

```
im
```

The imaginary unit.

Examples

source

`Base.MathConstants.pi` – Constant.

```
π

pi
```

The constant pi.

Examples

source

`Base.MathConstants.` – Constant.

```
e
```

The constant .

Examples

source

`Base.MathConstants.catalan` – Constant.

```
catalan
```

Catalan's constant.

Examples

source

Base.MathConstants.eulergamma – Constant.

```
γ

eulergamma
```

Euler's constant.

Examples

source

Base.MathConstants.golden – Constant.

```
φ

golden
```

The golden ratio.

Examples

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

source

`Base.isinf` – Function.

> isinf(f) -> Bool

Test whether a number is infinite.

source

`Base.isnan` – Function.

> isnan(f) -> Bool

Test whether a floating point number is not a number (NaN).

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

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

source

`Base.nextfloat` – Function.

> `nextfloat(x::IEEEFloat, 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

> 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

> 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

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

See `RoundingMode` for available rounding modes.

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

source

`Base.MPFR.BigFloat` – Method.

```
BigFloat(x)
```

Create an arbitrary precision floating point number. x may be an `Integer`, a `Float64` or a `BigInt`. The usual mathematical operators are defined for this type, and results are promoted to a `BigFloat`.

Note that because decimal literals are converted to floating point numbers when parsed, `BigFloat(2.1)` may not yield what you expect. You may instead prefer to initialize constants from strings via `parse`, or using the `big` string literal.

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`.

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.

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).

source

Integers

Base.count_ones – Function.

```
count_ones(x::Integer) -> Integer
```

Number of ones in the binary representation of x.

Examples

source

Base.count_zeros – Function.

```
count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of x.

Examples

source

Base.leading_zeros – Function.

```
leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of x.

Examples

source

Base.leading_ones – Function.

```
leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of x.

Examples

source

Base.trailing_zeros – Function.

```
trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of x.

Examples

source

Base.trailing_ones – Function.

```
trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of x.

Examples

source

Base.isodd – Function.

```
isodd(x::Integer) -> Bool
```

Return `true` if x is odd (that is, not divisible by 2), and `false` otherwise.

Examples

source

Base.iseven – Function.

```
iseven(x::Integer) -> Bool
```

Return `true` is x is even (that is, divisible by 2), and `false` otherwise.

Examples

source

Base.@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

Base.@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

45.4 BigFloats

The BigFloat type implements arbitrary-precision floating-point arithmetic using the GNU MPFR library.

Base.precision – Function.

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

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.

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`

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, prec::Int)
```

Create a representation of x as a BigFloat with precision `prec`.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, rounding::RoundingMode)
```

Create a representation of x as a BigFloat with the current global precision and Rounding Mode rounding.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, prec::Int, rounding::RoundingMode)
```

Create a representation of x as a BigFloat with precision prec and Rounding Mode rounding.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x)
```

Create an arbitrary precision floating point number. x may be an Integer, a Float64 or a BigInt. The usual mathematical operators are defined for this type, and results are promoted to a BigFloat.

Note that because decimal literals are converted to floating point numbers when parsed, BigFloat(2.1) may not yield what you expect. You may instead prefer to initialize constants from strings via parse, or using the big string literal.

source

```
BigFloat(x::String)
```

Create a representation of the string x as a BigFloat.

source

Base.@big_str – Macro.

```
@big_str str
@big_str(str)
```

@big_str parses a string into a BigInt Throws an ArgumentError if the string is not a valid integer Removes all underscores _ from the string

source

Chapter 46

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)
```

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. Without 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

source

Base.sizeof – Method.

```
sizeof(str::AbstractString)
```

Size, in bytes, of the string s. Equal to the number of code units in s multiplied by the size, in bytes, of one code unit in s.

Examples

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

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

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.

source

```
string(xs...)
```

Create a string from any values using the `print` function.

Examples

source

Base.repeat – Method.

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string r times. This can be written as `s^r`.

See also: `^`

Examples

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

source

Base.repr – Method.

```
repr(x; context=nothing)
```

Create a string from any value using the `show` function.

The optional keyword argument `context` can be set to an `I0` 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

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

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).

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)`.

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.

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

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.

Examples

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.

> 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`.

> 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

> 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`.

Examples

> 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}`.

Examples

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

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

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

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

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

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

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

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

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

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:end` such that `string[start:end] == "substring"`, or `nothing` if unmatched.

Examples

source

Base.findlast – Method.

```
findlast(pattern::AbstractString, string::AbstractString)
findlast(pattern::Regex, string::String)
```

Find the last occurrence of `pattern` in `string`. Equivalent to `findlast(pattern, string, lastindex(s))`.

Examples

source

Base.findprev – Method.

```
findprev(pattern::AbstractString, string::AbstractString, start::Integer)
findprev(pattern::Regex, string::String, start::Integer)
```

Find the previous 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[findprev(x, s, i)] == x`:

`findprev("substring", string, i)` = `start:end` such that `string[start:end] == "substring"`, or `nothing` if unmatched.

Examples

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

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

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

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

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

source

Base.strip – Function.

```
strip(str::AbstractString, [chars])
```

Remove leading and trailing characters from `str`.

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, or a predicate function.

Examples

source

Base.lstrip – Function.

```
lstrip([pred=isspace,] str::AbstractString)
lstrip(str::AbstractString, chars)
```

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

source

Base.rstrip – Function.

```
rstrip([pred=isspace,] str::AbstractString)
rstrip(str::AbstractString, chars)
```

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 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

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

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

source

Base.first – Method.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

source

Base.last – Method.

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last n characters of s.

source

Base.Unicode.uppercase – Function.

```
uppercase(s::AbstractString)
```

Return s with all characters converted to uppercase.

Examples

source

Base.Unicode.lowercase – Function.

```
lowercase(s::AbstractString)
```

Return s with all characters converted to lowercase.

Examples

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

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

source

`Base.Unicode.lowercasefirst` – Function.

```
lowercasefirst(s::AbstractString)
```

Return s with the first character converted to lowercase.

See also: `uppercasefirst`, `uppercase`, `lowercase`, `titlecase`

Examples

source

`Base.join` – Function.

```
join([io::IO,] strings, delim, [last])
```

Join an array of `strings` into a single string, inserting the given delimiter 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`. For example,

Examples

`strings` can be any iterable over elements x which are convertible to strings via `print(io::IOBuffer, x)`. `strings` will be printed to `io`.

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

source

`Base.chomp` – Function.

```
chomp(s::AbstractString)
```

Remove a single trailing newline from a string.

Examples

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

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

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

source

`Base.Unicode.textwidth` – Function.

```
textwidth(c)
```

Give the number of columns needed to print a character.

Examples

source

```
textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

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

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

source

Base.Unicode.isdigit – Function.

```
isdigit(c::AbstractChar) -> Bool
```

Tests whether a character is a decimal digit (0-9).

Examples

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

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

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

source

Base.Unicode.isprint – Function.

```
isprint(c::AbstractChar) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

Examples

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

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

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

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

source

`Core.Symbol` – Type.

```
Symbol(x...) -> Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

Examples

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

See also

unescape_string for the reverse operation.

source

`Base.unescape_string` – Function.

```
unescape_string(str::AbstractString)::AbstractString
unescape_string(io, str::AbstractString)::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 following escape sequences are recognised:

- Escaped backslash (\\)
- Escaped double-quote (\")
- Standard C escape sequences (\a, \b, \t, \n, \v, \f, \r, \e)
- Unicode code points (\u or \U prefixes with 1-4 trailing hex digits)
- Hex bytes (\x with 1-2 trailing hex digits)

- Octal bytes (\ with 1-3 trailing octal digits)

Examples

See also

escape_string.

source

Chapter 47

Arrays

47.1 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

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

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

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

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

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

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

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

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

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

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

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}
```

An N dimensional *strided* array with elements of type T. These arrays follow the strided array interface. 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

source

Base.zeros – Function.

```
zeros([T=Float64,] dims...)
```

Create an Array, with element type T, of all zeros with size specified by dims. See also fill, ones.

Examples

source

Base.ones – Function.

```
ones([T=Float64,] dims...)
```

Create an `Array`, with element type T, of all ones with size specified by `dims`. See also: `fill`, `zeros`.

Examples

source

Base.BitArray – Type.

```
BitArray{N} <: DenseArray{Bool, N}
```

Space-efficient N-dimensional boolean array, which stores one bit per boolean value.

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

source

Base.BitArray – Method.

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

source

Base.trues – Function.

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

Examples

source

Base.falses – Function.

```
falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

source

Base.fill – Function.

```
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`.

Examples

If x is an object reference, all elements will refer to the same object. `fill(Foo(), dims)` will return an array filled with the result of evaluating `Foo()` once.

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

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:

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArrays` are both mutable and can support 1-dimensional arrays:

Since `BitArrays` can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

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

47.2   Basic functions

Base.ndims – Function.

```
ndims(A::AbstractArray) -> Integer
```

Return the number of dimensions of A.

Examples

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

source

Base.axes – Method.

```
axes(A)
```

Return the tuple of valid indices for array A.

Examples

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

source

Base.length – Method.

```
length(A::AbstractArray)
```

Return the number of elements in the array, defaults to `prod(size(A))`.

Examples

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 $1 : \text{length}(A)$. 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, `eachindex` will return an iterable that spans the largest range along each dimension.

Examples

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 or cartesian indexing. If you decide to 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) convert 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.conj!` – Function.

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also `conj`.

Examples

source

`Base.stride` – Function.

```
stride(A, k::Integer)
```

Return the distance in memory (in number of elements) between adjacent elements in dimension k.

Examples

source

`Base.strides` – Function.

```
strides(A)
```

Return a tuple of the memory strides in each dimension.

Examples

source

47.3   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 Numbers, Strings, Symbols, Types, Functions 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

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)`.

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

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.broadcast_axes` – Function.

```
Base.broadcast_axes(A)
```

Compute the axes for A.

This should only be specialized for objects that do not define axes but want to participate in broadcasting.

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 `AbstractArrayStyles` 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

source

47.4 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

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

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

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

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

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:

For cartesian to linear index conversion, see `LinearIndices`.

source

Base.Dims – Type.

```
Dims{N}
```

An `NTuple` of N Ints 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 `AbstractVectors` 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

```
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:

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

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

source

47.5   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

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.

Examples

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.

Examples

source

Base.parent – Function.

```
parent(A)
```

Returns the "parent array" of an array view type (e.g., SubArray), or the array itself if it is not a view.

Examples

source

Base.parentindices – Function.

```
parentindices(A)
```

From an array view A, returns the corresponding indices in the parent.

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

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

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

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

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

See also reshape.

source

47.6   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

source

Base.hcat – Function.

```
hcat(A...)
```

Concatenate along dimension 2.

Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

source

47.7   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 inital value `init` may optionally be privided 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

Examples

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

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

source

```
cumprod(x::AbstractVector)
```

Cumulative product of a vector. 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

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

source

```
cumsum(x::AbstractVector)
```

Cumulative sum a vector. 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

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::AbstractMatrix; dims::Integer)
```

Finite difference operator of matrix or vector A. If A is a matrix, specify the dimension over which to operate with the dims keyword argument.

Examples

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

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

source

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string r times. This can be written as s^r.

See also: ^

Examples

source

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character r times. This can equivalently be accomplished by calling c^r.

Examples

source

Base.rot180 – Function.

```
rot180(A)
```

Rotate matrix A 180 degrees.

Examples

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

source

Base.rotl90 – Function.

```
rotl90(A)
```

Rotate matrix A left 90 degrees.

Examples

source

```
rotl90(A, k)
```

Rotate matrix A left 90 degrees an integer k number of times. If k is zero or a multiple of four, this is equivalent to a copy.

Examples

source

Base.rotr90 – Function.

```
rotr90(A)
```

Rotate matrix A right 90 degrees.

Examples

source

```
rotr90(A, k)
```

Rotate matrix A right 90 degrees an integer k number of times. If k is zero or a multiple of four, this is equivalent to a copy.

Examples

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

source

47.8   Combinatorics

Base.invperm – Function.

```
invperm(v)
```

Return the inverse permutation of v. If B = A[v], then A == B[invperm(v)].

Examples

source

Base.isperm – Function.

```
isperm(v) -> Bool
```

Return true if v is a valid permutation.

Examples

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

source

Base.invpermute! – Function.

```
invpermute!(v, p)
```

Like permute!, but the inverse of the given permutation is applied.

Examples

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

source

```
reverse(A; dims::Integer)
```

Reverse A in dimension dims.

Examples

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

source

`Base.reverse!` – Function.

```
reverse!(v [, start=1 [, stop=length(v) ]]) -> v
```

In-place version of `reverse`.

Examples

source

Chapter 48

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

In this example, b is a runnable `Task` that hasn't started yet.

> 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

> source

`Base.istaskstarted` – Function.

> ```
> istaskstarted(t::Task) -> Bool
> ```

Determine whether a task has started executing.

Examples

> source

`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.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

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` type does this, and so can be used for level-triggered events.

> source

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

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

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.Channel` – Type.

```
Channel{T}(sz::Int)
```

Constructs a `Channel` with an internal buffer that can hold a maximum of `sz` 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(Inf)`: equivalent to `Channel{Any}(typemax(Int))`
- `Channel(sz)`: equivalent to `Channel{Any}(sz)`

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.

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)
```

Close a channel. An exception 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

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 invoca-
tion, 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

Chapter 49

Multi-Threading

This experimental interface supports Julia's multi-threading capabilities. Types and functions described here might (and likely will) change in the future.

`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

`Base.Threads.@threads` – Macro.

```
Threads.@threads
```

A macro to parallelize a for-loop to run with multiple threads. This spawns `nthreads()` number of threads, splits the iteration space amongst them, and iterates in parallel. A barrier is placed at the end of the loop which waits for all the threads to finish execution, and the loop returns.

source

`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

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

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

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

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

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

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

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

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

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

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

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

49.1   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

49.2   Synchronization Primitives

`Base.Threads.AbstractLock` – Type.

> ```
> AbstractLock
> ```

Abstract supertype describing types that implement the thread-safe 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

`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 reentrant 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`.

This lock is NOT threadsafe. See `Threads.Mutex` for a threadsafe lock.

> source

`Base.Threads.Mutex` – Type.

```
Mutex()
```

These are standard system mutexes for locking critical sections of logic.

On Windows, this is a critical section object, on pthreads, this is a `pthread_mutex_t`.

See also `SpinLock` for a lighter-weight lock.

> source

`Base.Threads.SpinLock` – Type.

> SpinLock()

Create a non-reentrant lock. Recursive use will result in a deadlock. 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, perhaps a lock is the wrong way to synchronize.

See also RecursiveSpinLock for a version that permits recursion.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

source

Base.Threads.RecursiveSpinLock – Type.

> RecursiveSpinLock()

Creates a reentrant lock. The same thread can acquire the lock as many times as required. Each lock must be matched with an unlock.

See also SpinLock for a slightly faster version.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

source

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.

This construct is NOT threadsafe.

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

Chapter 50

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.LOAD_PATH` – Constant.

> LOAD_PATH

An array of paths for `using` and `import` statements to consdier as project environments or package directories when loading code. See 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 51

Filesystem

`Base.Filesystem.pwd` – Function.

```
pwd() -> AbstractString
```

Get the current working directory.

Examples

source

`Base.Filesystem.cd` – Method.

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

Examples

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

source

`Base.Filesystem.readdir` – Function.

```
readdir(dir::AbstractString=".") -> Vector{String}
```

Return the files and directories in the directory `dir` (or the current working directory if not given).

Examples

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

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

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

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 modes (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. RawFDs 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 `dest`. `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. 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.

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

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

source

`Base.Filesystem.touch` – Function.

```
touch(path::AbstractString)
```

Update the last-modified timestamp on a file to the current time. Return `path`.

Examples

We can see the `mtime` has been modified by `touch`.

source

`Base.Filesystem.tempname` – Function.

```
tempname()
```

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.

> **Warning**
>
> This can lead to race conditions if another process obtains the same file name and creates the file before you are able to. Using `mktemp()` is recommended instead.

`Base.Filesystem.tempdir` – Function.

```
tempdir()
```

Obtain the path of a temporary directory (possibly shared with other processes).

`Base.Filesystem.mktemp` – Method.

```
mktemp(parent=tempdir())
```

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.

`Base.Filesystem.mktemp` – Method.

```
mktemp(f::Function, parent=tempdir())
```

Apply the function f to the result of `mktemp(parent)` and remove the temporary file upon completion.

`Base.Filesystem.mktempdir` – Method.

```
mktempdir(parent=tempdir())
```

Create a temporary directory in the `parent` directory and return its path. If `parent` does not exist, throw an error.

`Base.Filesystem.mktempdir` – Method.

```
mktempdir(f::Function, parent=tempdir())
```

Apply the function f to the result of `mktempdir(parent)` and remove the temporary directory upon completion.

`Base.Filesystem.isblockdev` – Function.

```
isblockdev(path) -> Bool
```

Return `true` if `path` is a block device, `false` otherwise.

`Base.Filesystem.ischardev` – Function.

```
ischardev(path) -> Bool
```

Return `true` if `path` is a character device, `false` otherwise.

`Base.Filesystem.isdir` – Function.

```
isdir(path) -> Bool
```

Return `true` if `path` is a directory, `false` otherwise.

Examples

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

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 `path` is a valid filesystem path, `false` otherwise.

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() -> AbstractString
```

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.

Examples

See also: basename

source

`Base.Filesystem.basename` – Function.

```
basename(path::AbstractString) -> AbstractString
```

Get the file name part of a path.

Examples

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

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

source

`Base.Filesystem.joinpath` – Function.

```
joinpath(parts...) -> AbstractString
```

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.

Examples

source

`Base.Filesystem.abspath` – Function.

```
abspath(path::AbstractString) -> AbstractString
```

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...) -> AbstractString
```

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) -> AbstractString
```

Normalize a path, removing "." and ".." entries.

Examples

source

Base.Filesystem.realpath – Function.

```
realpath(path::AbstractString) -> AbstractString
```

Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

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

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

source

Chapter 52

I/O and Network

52.1 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(filename::AbstractString; 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.

> source

```
open(filename::AbstractString, [mode::AbstractString]) -> 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`

Examples

source

```
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

source

```
open(command, stdio=devnull; write::Bool = false, read::Bool = !write)
```

Start running command asynchronously, and return a tuple (`stream`, `process`). If read is true, then `stream` reads from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `write` is true, then `stream` writes to the process's standard input and `stdio` optionally specifies the process's standard output stream.

source

```
open(f::Function, command, mode::AbstractString="r", stdio=devnull)
```

Similar to `open(command, mode, stdio)`, 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.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

source

```
IOBuffer(string::String)
```

Create a read-only `IOBuffer` on the data underlying the given string.

Examples

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

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).

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

source

Base.read – Function.

```
read(io::IO, T)
```

Read a single value of type T from io, in canonical binary representation.

```
read(io::IO, String)
```

Read the entirety of io, as a String.

Examples

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

Base.read! – Function.

```
read!(stream::IO, array::Union{Array, BitArray})
read!(filename::AbstractString, array::Union{Array, BitArray})
```

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.

See read for a description of 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

source

Base.seek – Function.

```
seek(s, pos)
```

Seek a stream to the given position.

Examples

source

Base.seekstart – Function.

```
seekstart(s)
```

Seek a stream to its beginning.

Examples

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

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

source

Base.iswritable – Function.

```
iswritable(io) -> Bool
```

Return `true` if the specified IO object is writable (if that can be determined).

Examples

source

Base.isreadable – Function.

```
isreadable(io) -> Bool
```

Return `true` if the specified IO object is readable (if that can be determined).

Examples

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

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

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

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

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

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{UInt8,1}`.

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 small values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements.
- `: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

source

Base.IOContext – Method.

```
IOContext(io::IO, context::IOContext)
```

Create an `IOContext` that wraps an alternate `IO` but inherits the properties of `context`.

source

52.2   Text I/O

Base.show – Method.

```
show(x)
```

Write an informative text representation of a value to the current output stream. New types should overload `show(io, x)` where the first argument is a stream. The representation used by `show` generally includes Julia-specific formatting and type information.

source

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

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 of values `xs` if there is one, otherwise call `show`. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

Printing `nothing` is deprecated and will throw an error in the future.

Examples

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

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 suggersted (in bytes) to allocate for the buffer used to write the string.

Examples

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

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

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

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

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

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

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

To get your TTY size,

source

52.3   Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by over-loading `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.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).

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

Base.show – Method.

```
show(io, mime, x)
```

The `display` functions ultimately call show in order to write an object x as a given `mime` type to a given I/O stream io (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type T, it is only necessary to define a new show method for T, via: `show(io, ::MIME"mime", x::T) = ...`, where mime is a MIME-type string and the function body calls `write` (or similar) to write that representation of x to io. (Note that the `MIME""` notation only supports literal strings; to construct MIME types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `AbstractDisplay` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function show.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls show with 2 arguments. Therefore, this case should be handled by defining a 2-argument `show(io::IO, x::MyType)` method.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The first argument to show can be an `IOContext` specifying output format properties. See `IOContext` for details.

source

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

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

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

source

52.4 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

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 53

Punctuation

Extended documentation for mathematical symbols & functions is [here](#).

symbol	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^H
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)
a:b	range a, a+1, a+2, ..., b
a:s:b	range a, a+s, a+2s, ..., b
:	index an entire dimension (firstindex:lastindex), see Colon
::	type annotation or typeassert, depending on context
:( )	quoted expression
:a	symbol a
<:	subtype operator
>:	supertype operator (reverse of subtype operator)
===	egal comparison operator

Chapter 54

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:

You can easily sort in reverse order as well:

To sort an array in-place, use the "bang" version of the sort function:

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

Arrays can easily be sorted according to an arbitrary transformation of their values:

Or in reverse order by a transformation:

If needed, the sorting algorithm can be chosen:

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.

54.1   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

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

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.

Examples

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

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

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

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

54.2   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

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

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

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

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

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

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. If `initialized` is `false` (the default), ix is initialized to contain the values `1:length(ix)`.

source

54.3 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:

`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`:

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 55

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

source

Base.Iterators.zip – Function.

> `zip(iters...)`

For a set of iterable objects, return an iterable of tuples, where the `ith` tuple contains the `ith` component of each input iterable.

Examples

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 `ith` 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 `enumerate(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

source

Base.Iterators.rest – Function.

> rest(iter, state)

An iterator that yields the same elements as `iter`, but starting at the given `state`.

Examples

source

Base.Iterators.countfrom – Function.

> countfrom(start=1, step=1)

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

source

Base.Iterators.take – Function.

> take(iter, n)

An iterator that generates at most the first n elements of `iter`.

Examples

source

Base.Iterators.drop – Function.

> drop(iter, n)

An iterator that generates all but the first n elements of `iter`.

Examples

source

Base.Iterators.cycle – Function.

> cycle(iter)

An iterator that cycles through `iter` forever. If `iter` is empty, so is `cycle(iter)`.

Examples

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

source

Base.Iterators.product – Function.

```
product(iters...)
```

Return an iterator over the product of several iterators. Each generated element is a tuple whose `ith` element comes from the `ith` argument iterator. The first iterator changes the fastest.

Examples

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

source

`Base.Iterators.partition` – Function.

```
partition(collection, n)
```

Iterate over a collection n elements at a time.

Examples

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

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

source

Chapter 56

C Interface

ccall – Keyword.

```
ccall((function_name, library), 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. 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 `closure` 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`.

See manual section on ccall and cfunction usage.

Examples

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 ith 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 ith 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, 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

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.

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.

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, iftrue)
```

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.

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. Cstrings 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_ts. Cwstrings 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 57

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 58

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

Chapter 59

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::Union{Ptr{Cvoid}, UInt}) -> 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 `StackFrame`s) 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 60

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:

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:

However, since the automatic vectorization cannot be relied upon, future use will mostly be via libraries that use llvmcall.

Part V

Standard Library

Chapter 61

Base64

– 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

source

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.

source

Base64.Base64DecodePipe – Type.

```
Base64DecodePipe(istream)
```

Return a new read-only I/O stream, which decodes base64-encoded data read from `istream`.

Examples

source

Base64.base64decode – Function.

```
base64decode(string)
```

Decode the base64-encoded `string` and returns a `Vector{UInt8}` of the decoded bytes.

See also `base64encode`.

Examples

source

`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`.

source

Chapter 62

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.

source

– 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.

source

Chapter 63

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.

63.1 Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

`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

---

[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.

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.

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.

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

63.2   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).

63.3   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:

While propercase return the same value in the corresponding `Period` type:

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

One may also access the underlying `UTInstant` or integer value:

63.4   Query Functions

Query functions provide calendrical information about a `TimeType`. They include information about the day of the week:

Month of the year:

As well as information about the `TimeType`'s year and quarter:

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:

The above mentioned functions can then be used to perform the queries:

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr` will error.

63.5   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:

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:

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:

63.6   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.

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:

This is useful with the do-block syntax for more complex temporal expressions:

The `Base.filter` method can be used to obtain all valid dates/moments in a specified range:

Additional examples and tests are available in `stdlib/Dates/test/adjusters.jl`.

63.7 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.

63.8 Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`:

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?

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:

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:

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 64

API reference

64.1 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.

source

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.

source

Dates.Instant – Type.

```
Instant
```

Instant types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

source

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.

> source

`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`.

> source

`Dates.DateTime` – Type.

> DateTime

`DateTime` wraps a `UTInstant{Millisecond}` and interprets it according to the proleptic Gregorian calendar.

> source

`Dates.Date` – Type.

> Date

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

> source

`Dates.Time` – Type.

> Time

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

> source

64.2 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`.

> source

`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)`.

> source

`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

source

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.

source

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.

source

Dates.format – Function.

```
format(io::IO, tok::AbstractDateToken, dt::TimeType, locale)
```

Format the `tok` token from `dt` and write it to `io`. The formatting can be based on `locale`.

All subtypes of `AbstractDateToken` must define this method in order to be able to print a Date / DateTime object according to a `DateFormat` containing that token.

source

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:

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".

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.

source

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
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
yyyymmdd	19960101	Matches fixed-width year, month, and day

`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.

source

`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.

source

`Dates.Date` – Method.

```
Date(y, [m, d]) -> Date
```

Construct a `Date` type by parts. Arguments must be convertible to `Int64`.

source

`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)`.

source

`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

source

`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.

source

`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.

source

`Dates.Date` – Method.

```
Date(d::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string `d` using a `DateFormat` object `df`.

source

`Dates.Time` – Method.

```
Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a `Time` type by parts. Arguments must be convertible to `Int64`.

source

`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)`.

source

`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

source

`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.

source

`Dates.now` – Method.

```
now() -> DateTime
```

Return a `DateTime` corresponding to the user's system time including the system timezone locale.

source

`Dates.now` – Method.

```
now(::Type{UTC}) -> DateTime
```

Return a `DateTime` corresponding to the user's system time as UTC/GMT.

source

`Base.eps` – Function.

```
eps(::DateTime) -> Millisecond
eps(::Date) -> Day
eps(::Time) -> Nanosecond
```

Returns `Millisecond(1)` for `DateTime` values, `Day(1)` for `Date` values, and `Nanosecond(1)` for `Time` values.

source

Accessor Functions

`Dates.year` – Function.

```
year(dt::TimeType) -> Int64
```

The year of a `Date` or `DateTime` as an `Int64`.

source

`Dates.month` – Function.

```
month(dt::TimeType) -> Int64
```

The month of a `Date` or `DateTime` as an `Int64`.

source

`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

source

`Dates.day` – Function.

```
day(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

source

`Dates.hour` – Function.

```
hour(dt::DateTime) -> Int64
```

The hour of day of a `DateTime` as an `Int64`.

source

```
hour(t::Time) -> Int64
```

The hour of a `Time` as an `Int64`.

source

`Dates.minute` – Function.

```
minute(dt::DateTime) -> Int64
```

The minute of a `DateTime` as an `Int64`.

source

```
minute(t::Time) -> Int64
```

The minute of a `Time` as an `Int64`.

source

`Dates.second` – Function.

```
second(dt::DateTime) -> Int64
```

The second of a `DateTime` as an `Int64`.

source

```
second(t::Time) -> Int64
```

The second of a `Time` as an Int64.

> source

Dates.millisecond – Function.

> ```
> millisecond(dt::DateTime) -> Int64
> ```

The millisecond of a `DateTime` as an Int64.

> source

> ```
> millisecond(t::Time) -> Int64
> ```

The millisecond of a `Time` as an Int64.

> source

Dates.microsecond – Function.

> ```
> microsecond(t::Time) -> Int64
> ```

The microsecond of a `Time` as an Int64.

> source

Dates.nanosecond – Function.

> ```
> nanosecond(t::Time) -> Int64
> ```

The nanosecond of a `Time` as an Int64.

> source

Dates.Year – Method.

> ```
> Year(v)
> ```

Construct a `Year` object with the given v value. Input must be losslessly convertible to an Int64.

> source

Dates.Month – Method.

> ```
> Month(v)
> ```

Construct a `Month` object with the given v value. Input must be losslessly convertible to an Int64.

> source

Dates.Week – Method.

> ```
> Week(v)
> ```

Construct a `Week` object with the given v value. Input must be losslessly convertible to an Int64.

> source

Dates.Day – Method.

> ```
> Day(v)
> ```

Construct a Day object with the given v value. Input must be losslessly convertible to an `Int64`.

source

Dates.Hour – Method.

```
Hour(dt::DateTime) -> Hour
```

The hour part of a DateTime as a `Hour`.

source

Dates.Minute – Method.

```
Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a `Minute`.

source

Dates.Second – Method.

```
Second(dt::DateTime) -> Second
```

The second part of a DateTime as a `Second`.

source

Dates.Millisecond – Method.

```
Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a `Millisecond`.

source

Dates.Microsecond – Method.

```
Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a `Microsecond`.

source

Dates.Nanosecond – Method.

```
Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a `Nanosecond`.

source

Dates.yearmonth – Function.

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a Date or DateTime.

source

Dates.monthday – Function.

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a `Date` or `DateTime`.

source

Dates.yearmonthday – Function.

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a `Date` or `DateTime`.

source

Query Functions

Dates.dayname – Function.

```
dayname(dt::TimeType; locale="english") -> String
```

Return the full day name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`.

Examples

source

Dates.dayabbr – Function.

```
dayabbr(dt::TimeType; locale="english") -> String
```

Return the abbreviated name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`.

Examples

source

Dates.dayofweek – Function.

```
dayofweek(dt::TimeType) -> Int64
```

Return the day of the week as an `Int64` with 1 = Monday, 2 = Tuesday, etc..

Examples

source

Dates.dayofmonth – Function.

```
dayofmonth(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

source

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

"'jldoctest julia> Dates.dayofweekofmonth(Date("2000-02-01")) 1

julia> Dates.dayofweekofmonth(Date("2000-02-08")) 2

julia> Dates.dayofweekofmonth(Date("2000-02-15")) 3 ""

source

`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

source

`Dates.monthname` – Function.

```
monthname(dt::TimeType; locale="english") -> String
```

Return the full name of the month of the `Date` or `DateTime` in the given `locale`.

Examples

source

`Dates.monthabbr` – Function.

```
monthabbr(dt::TimeType; locale="english") -> String
```

Return the abbreviated month name of the `Date` or `DateTime` in the given `locale`.

Examples

source

`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

source

`Dates.isleapyear` – Function.

```
isleapyear(dt::TimeType) -> Bool
```

Return `true` if the year of `dt` is a leap year.

Examples

source

`Dates.dayofyear` – Function.

```
dayofyear(dt::TimeType) -> Int
```

Return the day of the year for `dt` with January 1st being day 1.

source

`Dates.daysinyear` – Function.

```
daysinyear(dt::TimeType) -> Int
```

Return 366 if the year of `dt` is a leap year, otherwise return 365.

Examples

source

`Dates.quarterofyear` – Function.

```
quarterofyear(dt::TimeType) -> Int
```

Return the quarter that `dt` resides in. Range of value is 1:4.

source

`Dates.dayofquarter` – Function.

```
dayofquarter(dt::TimeType) -> Int
```

Return the day of the current quarter of `dt`. Range of value is 1:92.

source

Adjuster Functions

`Base.trunc` – Method.

```
trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

source

`Dates.firstdayofweek` – Function.

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

source

`Dates.lastdayofweek` – Function.

```
lastdayofweek(dt::TimeType) -> TimeType
```

Adjusts dt to the Sunday of its week.

Examples

source

Dates.firstdayofmonth – Function.

```
firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts dt to the first day of its month.

Examples

source

Dates.lastdayofmonth – Function.

```
lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts dt to the last day of its month.

Examples

source

Dates.firstdayofyear – Function.

```
firstdayofyear(dt::TimeType) -> TimeType
```

Adjusts dt to the first day of its year.

Examples

source

Dates.lastdayofyear – Function.

```
lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts dt to the last day of its year.

Examples

source

Dates.firstdayofquarter – Function.

```
firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts dt to the first day of its quarter.

Examples

source

Dates.lastdayofquarter – Function.

```
lastdayofquarter(dt::TimeType) -> TimeType
```

Adjusts dt to the last day of its quarter.

Examples

source

`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.

source

`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.

source

`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.

source

`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.

source

`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.

source

`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.

source

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`.

source

Dates.CompoundPeriod – Method.

```
CompoundPeriod(periods) -> CompoundPeriod
```

Construct a `CompoundPeriod` from a `Vector` of `Periods`. All `Periods` of the same type will be added together.

Examples

source

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.

source

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))`.

source

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))`.

source

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))`.

Valid rounding modes for `round(::TimeType, ::Period, ::RoundingMode)` are RoundNearestTiesUp (default), RoundDown (floor), and RoundUp (ceil).

  source

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))`.

Rounding to a `precision` of Months or Years is not supported, as these `Periods` are of inconsistent length.

  source

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))`.

Rounding to a `precision` of Months or Years is not supported, as these `Periods` are of inconsistent length.

  source

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))`.

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are RoundNearestTiesUp (default), RoundDown (floor), and RoundUp (ceil).

Rounding to a `precision` of Months or Years is not supported, as these `Periods` are of inconsistent length.

  source

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.

source

```
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.

source

`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`.

source

`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`.

source

`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`.

source

`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`.

source

Conversion Functions

`Dates.today` – Function.

```
today() -> Date
```

Return the date portion of `now()`.

source

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`.

source

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`.

source

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`.

source

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`.

source

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`.

source

Dates.datetime2rata – Function.

```
datetime2rata(dt::TimeType) -> Int64
```

Return the number of Rata Die days since epoch from the given `Date` or `DateTime`.

source

Constants

Days of the Week:

Months of the Year:

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

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 65

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

```
source
```

`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.

```
source
```

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, delim::AbstractChar, T::Type; options...)
```

The end of line delimiter is taken as \n.

Examples

source

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

source

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

source

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

source

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

source

Chapter 66

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.

source

```
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`.
- `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.

source

```
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.

source

```
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)`.

source

`Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

source

`Distributed.nworkers` – Function.

```
nworkers()
```

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

source

Distributed.procs – Method.

```
procs()
```

Return a list of all process identifiers.

source

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.

source

Distributed.workers – Function.

```
workers()
```

Return a list of all worker process identifiers.

source

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.

source

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.

source

```
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.

source

Distributed.myid – Function.

```
myid()
```

Get the id of the current process.

source

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:

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.

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.

source

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.

source

Distributed.Future – Type.

```
Future(pid::Integer=myid())
```

Create a Future on process pid. The default pid is the current process.

source

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.

source

Base.wait – Function.

```
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, the exception is propagated (re-thrown in the task that called `wait`).
- `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

```
wait(r::Future)
```

Wait for a value to become available for the specified future.

source

```
wait(r::RemoteChannel, args...)
```

Wait for a value to become available on the specified remote channel.

source

Base.fetch – Method.

```
fetch(x)
```

Waits and fetches a value from `x` depending on the type of `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.

- RemoteChannel: Wait for and get the value of a remote reference. Exceptions raised are same as for a Future.

Does not remove the item fetched.

source

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.

source

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.

source

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.

source

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 remotecalls 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.

source

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.

source

Base.put! – Method.

```
put!(rr::Future, v)
```

Store a value to a Future rr. Futures are write-once remote references. A put! on an already set Future throws an Exception. All asynchronous remote calls return Futures and set the value to the return value of the call upon completion.

source

Base.take! – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a RemoteChannel rr, removing the value(s) in the process.

source

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.

source

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:

```
c = Channel(1)
@async put!(c, remotecall_fetch(long_computation, p))
isready(c)  # will not block
```

source

Distributed.WorkerPool – Type.

```
WorkerPool(workers::Vector{Int})
```

Create a WorkerPool from a vector of worker ids.

source

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

The above would transfer `foo` only once to each worker.

source

`Distributed.default_worker_pool` – Function.

```
default_worker_pool()
```

`WorkerPool` containing idle `workers` - used by `remote(f)` and `pmap` (by default).

source

`Distributed.clear!` – Method.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

source

`Distributed.remote` – Function.

```
remote([::AbstractWorkerPool], f) -> Function
```

Return an anonymous function that executes function `f` on an available worker using `remotecall_fetch`.

source

`Distributed.remotecall` – Method.

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall` on it.

source

`Distributed.remotecall_wait` – Method.

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_wait` on it.

source

`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.

source

Distributed.remote_do – Method.

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

WorkerPool variant of `remote_do(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remote_do` on it.

source

Base.timedwait – Function.

```
timedwait(testcb::Function, secs::Float64; pollint::Float64=0.1)
```

Waits until `testcb` returns `true` or for `secs` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds.

source

Distributed.@spawn – Macro.

```
@spawn
```

Create a closure around an expression and run it on an automatically-chosen process, returning a `Future` to the result.

Examples

source

Distributed.@spawnat – Macro.

```
@spawnat
```

Create a closure around an expression and run the closure asynchronously on process p. Return a `Future` to the result. Accepts two arguments, p and an expression.

Examples

source

Distributed.@fetch – Macro.

```
@fetch
```

Equivalent to `fetch(@spawn expr)`. See `fetch` and `@spawn`.

Examples

source

Distributed.@fetchfrom – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See `fetch` and `@spawnat`.

Examples

source

Base.@async – Macro.

> @async

Wrap an expression in a Task and add it to the local machine's scheduler queue.

> 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

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
```

> source

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 processes.

Unlike @spawn and @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).

> source

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 `Symbols`. `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.

source

Distributed.remoteref_id – Function.

```
remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` 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.

source

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.

source

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.

source

Distributed.cluster_cookie – Method.

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

source

Distributed.cluster_cookie – Method.

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

source

66.1 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.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.

source

`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.

source

`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`.

source

`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.

source

[Distributed.init_worker]{.blue} – 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`.

source

[Distributed.start_worker]{.blue} – Function.

```
start_worker([out::IO=stdout], cookie::AbstractString=readline(stdin))
```

`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 closes stdin (after reading the cookie if required), redirects stderr to stdout, 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 does not return.

source

[Distributed.process_messages]{.blue} – 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`.

source

Chapter 67

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.

source

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.

source

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.

source

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.

source

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.

source

Chapter 68

Interactive Utilities

`Base.Docs.apropos` – Function.

```
apropos(string)
```

Search through all documentation for a string, ignoring case.

source

`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.

source

`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

source

`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.

source

`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

source

`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.

source

`InteractiveUtils.edit` – Method.

```
edit(function, [types])
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. The editor can be changed by setting JULIA_EDITOR, VISUAL or EDITOR as an environment variable.

source

`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.

source

`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.

source

`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.

source

`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; verbose_linetable=false)
```

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. If the `verbose_linetable` keyword is set, the linetable will be printed in verbose mode, showing all available information (rather than applying the usual heuristics). 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.

> source

`InteractiveUtils.code_llvm` – Function.

> `code_llvm([io=stdout,], f, types)`

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. Set the `raw` keyword for the full IR. To dump the entire module that encapsulates the function, with debug info and metadata, set the `dump_module` keyword.

> source

`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` and `optimize` by putting them and their value before the function call, like this:

> `@code_llvm raw=true dump_module=true 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. `dump_module` prints the entire module that encapsulates the function, with debug info and metadata.

> source

`InteractiveUtils.code_native` – Function.

> `code_native([io=stdout,], f, types; syntax = :att)`

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.

> source

`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.

> source

`InteractiveUtils.clipboard` – Function.

```
clipboard(x)
```

Send a printed form of x to the operating system clipboard ("copy").

source

```
clipboard() -> AbstractString
```

Return a string with the contents of the operating system clipboard ("paste").

source

Chapter 69

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:

In particular, note that `LibGit2.free` should be called afterward on the Ref object.

> source

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.

    source

`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.

    source

`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).

    source

`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"`.

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.

source

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.

source

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.

source

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.

source

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.

source

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

source

```
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

source

LibGit2.GitRemoteAnon – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString) -> GitRemote
```

Look up a remote git repository using only its URL, not its name.

Examples

source

LibGit2.GitRepo – Type.

```
LibGit2.GitRepo(path::AbstractString)
```

Open a git repository at `path`.

source

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`).

source

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).

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.

source

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).

> source

LibGit2.GitSignature – Type.

```
LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

> source

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.

> source

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.

> source

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.

> source

LibGit2.IndexEntry – Type.

```
LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

> source

LibGit2.IndexTime – Type.

> LibGit2.IndexTime

Matches the `git_index_time` struct.

`source`

`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

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.

  source

`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).

  source

`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.

    source

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.

    source

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`.

source

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:

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:

Note that no call to `free` is required as the data is allocated by Julia.

source

LibGit2.TimeStruct – Type.

> LibGit2.TimeStruct

Time in a signature. Matches the `git_time` struct.

source

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`.

source

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

source

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

> **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.

source

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

source

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.

source

LibGit2.authors – Function.

```
authors(repo::GitRepo) -> Vector{Signature}
```

Return all authors of commits to the `repo` repository.

Examples

source

LibGit2.branch – Function.

```
branch(repo::GitRepo)
```

Equivalent to `git branch`. Create a new branch from the current HEAD.

source

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

source

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

source

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

source

```
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

source

[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.

source

```
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.

source

[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.

source

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

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.

source

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.

source

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.

source

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.

source

`LibGit2.credentials_cb` – Function.

C function pointer for `credentials_callback`

source

`LibGit2.default_signature` – Function.

Return signature object. Free it after use.

source

`LibGit2.delete_branch` – Function.

```
LibGit2.delete_branch(branch::GitReference)
```

Delete the branch pointed to by `branch`.

source

`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

Equivalent to `git diff --name-only --diff-filter=<filter> <branch1> <branch2>`.

source

`LibGit2.entryid` – Function.

```
entryid(te::GitTreeEntry)
```

Return the `GitHash` of the object to which `te` refers.

source

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.

source

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.

source

```
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>]`.

source

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

source

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

source

LibGit2.fetchhead_foreach_cb – Function.

C function pointer for `fetchhead_foreach_callback`

source

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.

source

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")`.

source

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

source

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

source

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).

source

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.

source

`LibGit2.features` – Function.

```
features()
```

Return a list of git features the current version of libgit2 supports, such as threading or using HTTPS or SSH.

source

`LibGit2.filename` – Function.

```
filename(te::GitTreeEntry)
```

Return the filename of the object on disk to which `te` refers.

source

`LibGit2.filemode` – Function.

```
filemode(te::GitTreeEntry) -> Cint
```

Return the UNIX filemode of the object on disk to which `te` refers as an integer.

source

`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.

source

`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

source

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

source

LibGit2.head – Function.

```
LibGit2.head(repo::GitRepo) -> GitReference
```

Return a `GitReference` to the current HEAD of `repo`.

source

```
head(pkg::AbstractString) -> String
```

Return current HEAD GitHash of the pkg repo as a string.

source

LibGit2.head! – Function.

```
LibGit2.head!(repo::GitRepo, ref::GitReference) -> GitReference
```

Set the HEAD of `repo` to the object pointed to by `ref`.

source

LibGit2.head_oid – Function.

```
LibGit2.head_oid(repo::GitRepo) -> GitHash
```

Lookup the object id of the current HEAD of git repository `repo`.

source

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.

source

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.

source

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

source

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.

source

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

source

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

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

source

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

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

source

`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.

source

`LibGit2.isset` – Function.

```
isset(val::Integer, flag::Integer)
```

Test whether the bits of `val` indexed by `flag` are set (1) or unset (0).

source

`LibGit2.iszero` – Function.

```
iszero(id::GitHash) -> Bool
```

Determine whether all hexadecimal digits of the given `GitHash` are zero.

source

`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.

source

`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

Here, map visits each commit using the `GitRevWalker` and finds its `GitHash`.

source

`LibGit2.mirror_callback` – Function.

Mirror callback function

Function sets `+refs/*:refs/*` refspecs and `mirror` flag for remote reference.

source

LibGit2.mirror_cb – Function.

    C function pointer for `mirror_callback`

    source

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.

    source

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.

    source

LibGit2.name – Function.

    ```
LibGit2.name(ref::GitReference)
```

    Return the full name of ref.

    source

    ```
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

    source

    ```
LibGit2.name(tag::GitTag)
```

    The name of tag (e.g. `"v0.5"`).

    source

`LibGit2.need_update` – Function.

```
need_update(repo::GitRepo)
```

Equivalent to `git update-index`. Return `true` if `repo` needs updating.

source

`LibGit2.objtype` – Function.

```
objtype(obj_type::Consts.OBJECT)
```

Return the type corresponding to the enum value.

source

`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`.

source

`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.

source

```
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`.

source

LibGit2.posixpath – Function.

```
LibGit2.posixpath(path)
```

Standardise the path string `path` to use POSIX separators.

source

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"])`.

source

```
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>]`.

source

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`.

source

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.

source

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

source

LibGit2.raw – Function.

```
raw(id::GitHash) -> Vector{UInt8}
```

Obtain the raw bytes of the `GitHash` as a vector of length 20.

source

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.

source

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
```

source

LibGit2.ref_list – Function.

```
LibGit2.ref_list(repo::GitRepo) -> Vector{String}
```

Get a list of all reference names in the `repo` repository.

source

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

source

LibGit2.remotes – Function.

> LibGit2.remotes(repo::GitRepo)

Return a vector of the names of the remotes of repo.

source

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).

source

LibGit2.reset – Function.

> reset(val::Integer, flag::Integer)

Unset the bits of val indexed by flag, returning them to 0.

source

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.

source

Updates some entries, determined by the pathspecs, in the index from the target commit tree.

source

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

source

git reset [<committish>] [–] <pathspecs>...

source

> 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

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

source

`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.

source

`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

This will return `(-1, 0)`.

source

`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

source

`LibGit2.shortname` – Function.

```
LibGit2.shortname(ref::GitReference)
```

Return a shortened version of the name of `ref` that's "human-readable".

source

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.

source

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:

Refer to the git config syntax documentation for more details.

source

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.

source

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).

source

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.

source

LibGit2.tag_delete – Function.

```
LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

source

LibGit2.tag_list – Function.

```
LibGit2.tag_list(repo::GitRepo) -> Vector{String}
```

Get a list of all tags in the git repository `repo`.

source

LibGit2.target – Function.

```
LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

source

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.

source

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.

source

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.

source

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.

source

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.

source

LibGit2.url – Function.

```
url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Examples

source

LibGit2.version – Function.

```
version() -> VersionNumber
```

Return the version of libgit2 in use, as a VersionNumber.

source

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.

source

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.

source

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.

source

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

source

LibGit2.UserPasswordCredential – Type.

Credential that support only user and password parameters

source

LibGit2.SSHCredential – Type.

SSH credential type

source

LibGit2.isfilled – Function.

```
isfilled(cred::AbstractCredential) -> Bool
```

Verifies that a credential is ready for use in authentication.

source

LibGit2.CachedCredentials – Type.

Caches credential information for re-use

source

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.

source

`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.

source

`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.

source

Chapter 70

Dynamic Linker

`Libdl.dlopen` – Function.

```
dlopen(libfile::AbstractString [, flags::Integer])
```

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.

> source

`Libdl.dlopen_e` – Function.

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to dlopen, except returns a NULL pointer instead of raising errors.

> source

`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.

source

`Libdl.dlsym` – Function.

```
dlsym(handle, sym)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

source

`Libdl.dlsym_e` – Function.

```
dlsym_e(handle, sym)
```

Look up a symbol from a shared library handle, silently return NULL pointer on lookup failure.

source

`Libdl.dlclose` – Function.

```
dlclose(handle)
```

Close shared library referenced by handle.

source

`Libdl.dlext` – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

source

`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.

source

`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.

source

Chapter 71

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. Basic operations, such as `tr`, `det`, and `inv` are all supported:

As well as other useful operations, such as finding eigenvalues or eigenvectors:

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:

Since A is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

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:

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:

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.

71.1   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.

Elementary operations

Legend:

Matrix factorizations

Legend:

Type	Description
Symmetric	Symmetric matrix
Hermitian	Hermitian matrix
UpperTriangular	Upper triangular matrix
LowerTriangular	Lower triangular matrix
Tridiagonal	Tridiagonal matrix
SymTridiagonal	Symmetric tridiagonal matrix
Bidiagonal	Upper/lower bidiagonal matrix
Diagonal	Diagonal matrix
UniformScaling	Uniform scaling operator

Matrix type	+	−	*	\	Other functions with optimized methods
Symmetric				MV	inv, sqrt, exp
Hermitian				MV	inv, sqrt, exp
UpperTriangular			MV	MV	inv, det
LowerTriangular			MV	MV	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	/

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

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:

71.2   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.

71.3   Standard Functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

Base.:* – Method.

Matrix type	LAPACK	eigen	eigvals	eigvecs	svd	svdvals
Symmetric	SY		ARI			
Hermitian	HE		ARI			
UpperTriangular	TR	A	A	A		
LowerTriangular	TR	A	A	A		
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

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 ilth through the ihth 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)

Type	Description
Cholesky	Cholesky factorization
CholeskyPivoted	Pivoted Cholesky factorization
LU	LU factorization
LUTridiagonal	LU factorization for Tridiagonal matrices
QR	QR factorization
QRCompactWY	Compact WY form of the QR factorization
QRPivoted	Pivoted QR factorization
Hessenberg	Hessenberg decomposition
Eigen	Spectral decomposition
SVD	Singular value decomposition
GeneralizedSVD	Generalized SVD

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

source

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

source

LinearAlgebra.dot – Function.

```
dot(x, y)
x   y
```

For any iterable containers x and y (including arrays of any dimension) of numbers (or any element type for which dot is defined), compute the dot product (or inner product or scalar product), i.e. the sum of `dot(x[i],y[i])`, as if they were vectors.

x   y (where   can be typed by tab-completing \cdot in the REPL) is a synonym for `dot(x, y)`.

Examples

source

```
dot(x, y)
x   y
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated. When the vectors have equal lengths, calling dot is semantically equivalent to `sum(dot(vx,vy) for (vx,vy) in zip(x, y))`.

Examples

source

LinearAlgebra.cross – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

source

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

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.

source

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)

LinearAlgebra.Diagonal – Type.

```
Diagonal(A::AbstractMatrix)
```

Construct a matrix from the diagonal of A.

Examples

source

```
Diagonal(V::AbstractVector)
```

Construct a matrix with V as its diagonal.

Examples

source

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

source

```
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

source

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).

Examples

source

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first sub/super-diagonal, of the symmetric matrix A.

Examples

source

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

source

```
Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix A.

Examples

source

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

Note that Supper will not be equal to Slower unless A is itself symmetric (e.g. if A == transpose(A)).

source

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

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.

source

LinearAlgebra.LowerTriangular – Type.

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the the matrix A.

Examples

source

`LinearAlgebra.UpperTriangular` – Type.

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the the matrix A.

Examples

source

`LinearAlgebra.UniformScaling` – Type.

```
UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator, $\lambda*I$. See also `I`.

Examples

source

`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

source

`LinearAlgebra.lu!` – Function.

```
lu!(A, pivot=Val(true); check = true) -> LU
```

Supported function	LU	LU{T,Tridiagonal{T}}
/		
\		
inv		
det		
logdet		
logabsdet		
size		

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

source

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.

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

source

```
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 PivotedCholesky 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.

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.

source

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

source

```
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.

source

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.

source

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.

source

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.

source

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.

source

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

source

LinearAlgebra.ldlt! – Function.

```
ldlt!(S::SymTridiagonal) -> LDLt
```

Same as `ldlt`, but saves space by overwriting the input S, instead of creating a copy.

Examples

source

LinearAlgebra.qr – Function.

```
qr(A, pivot=Val(false)) -> F
```

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.

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.

Examples

> **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.

source

LinearAlgebra.qr! – Function.

```
qr!(A, pivot=Val(false))
```

`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.

Examples

source

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.

source

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], as a lower trapezoidal matrix V and an upper triangular matrix T where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

such that v_i is the ith column of V, and au_i is the ith diagonal element of T.

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 square matrix with `min(m,n)` columns, whose upper triangular part gives the matrix T above (the subdiagonal elements are ignored).

  Note

  This format should not to be confused with the older *WY* representation [2].

source

`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.

[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

- 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.

source

`LinearAlgebra.lq!` – Function.

```
lq!(A) -> LQ
```

Compute the LQ factorization of A, using the input matrix as a workspace. See also `lq`.

source

`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)`.

Examples

source

`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`.

and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

Examples

source

---

[3] J R Bunch and L Kaufman, Some stable methods for calculating inertia

`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.

source

`LinearAlgebra.eigvals` – Function.

```
eigvals(A; permute::Bool=true, scale::Bool=true) -> 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 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.

Examples

source

For a scalar input, `eigvals` will return a scalar.

Example

source

```
eigvals(A, B) -> values
```

Computes the generalized eigenvalues of A and B.

Examples

source

```
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.

source

```
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.

source

`LinearAlgebra.eigvals!` – Function.

```
eigvals!(A; permute::Bool=true, scale::Bool=true) -> values
```

Same as `eigvals`, but saves space by overwriting the input A, instead of creating a copy. 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**
>
> The input matrix A will not contain its eigenvalues after `eigvals!` is called on it - A is used as a workspace.

Examples

source

```
eigvals!(A, B) -> 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

source

```
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.

source

```
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.

source

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

source

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

source

`LinearAlgebra.eigvecs` – Function.

```
eigvecs(A::SymTridiagonal[, eigvals]) -> Matrix
```

Return a matrix M whose columns are the eigenvectors of A. (The kth 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

source

```
eigvecs(A; permute::Bool=true, scale::Bool=true) -> Matrix
```

Return a matrix M whose columns are the eigenvectors of A. (The kth eigenvector can be obtained from the slice M[:, k].) The `permute` and `scale` keywords are the same as for `eigen`.

Examples

source

```
eigvecs(A, B) -> Matrix
```

Return a matrix M whose columns are the generalized eigenvectors of A and B. (The kth eigenvector can be obtained from the slice M[:, k].)

Examples

source

`LinearAlgebra.eigen` – Function.

```
eigen(A; permute::Bool=true, scale::Bool=true) -> 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 kth 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.

Examples

source

```
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 kth generalized eigenvector can be obtained from the slice F.`vectors[:, k]`.)

Iterating the decomposition produces the components F.`values` and F.`vectors`.

Examples

source

```
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 kth 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.

source

```
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 kth 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.

source

LinearAlgebra.eigen! – Function.

```
eigen!(A, [B])
```

Same as `eigen`, but saves space by overwriting the input A (and B), instead of creating a copy.

source

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` 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

source

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.

source

LinearAlgebra.schur! – Function.

```
schur!(A::StridedMatrix) -> F::Schur
```

Same as schur but uses the input argument A as workspace.

Examples

source

```
schur!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as schur but uses the input matrices A and B as workspace.

source

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

source

```
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.

source

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.

source

```
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.

source

`LinearAlgebra.ordschur!` – Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as `ordschur` but overwrites the factorization F.

source

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Same as `ordschur` but overwrites the factorization F.

source

`LinearAlgebra.svd` – Function.

```
svd(A; full::Bool = false) -> 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.

Examples

source

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of A and B, returning a `GeneralizedSVD` factorization object F, such that A = F.U*F.D1*F.R0*F.Q' and B = F.V*F.D2*F.R0*F.Q'.

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

source

`LinearAlgebra.svd!` – Function.

```
svd!(A; full::Bool = false) -> SVD
```

`svd!` is the same as svd, but saves space by overwriting the input A, instead of creating a copy.

Examples

source

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as svd, but modifies the arguments A and B in-place, instead of making copies.

Examples

source

`LinearAlgebra.svdvals` – Function.

```
svdvals(A)
```

Return the singular values of A in descending order.

Examples

source

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B. See also svd.

Examples

source

`LinearAlgebra.svdvals!` – Function.

```
svdvals!(A)
```

Return the singular values of A, saving space by overwriting the input. See also svdvals and svd.

Examples

source

```
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.

Examples

source

`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`

source

`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`

source

```
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`

source

```
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`

source

`LinearAlgebra.triu` – Function.

```
triu(M)
```

Upper triangle of a matrix.

Examples

source

```
triu(M, k::Integer)
```

Returns the upper triangle of M starting from the kth superdiagonal.

Examples

source

`LinearAlgebra.triu!` – Function.

```
triu!(M)
```

Upper triangle of a matrix, overwriting M in the process. See also `triu`.

source

```
triu!(M, k::Integer)
```

Return the upper triangle of M starting from the kth superdiagonal, overwriting M in the process.

Examples

source

`LinearAlgebra.tril` – Function.

```
tril(M)
```

Lower triangle of a matrix.

Examples

source

```
tril(M, k::Integer)
```

Returns the lower triangle of M starting from the kth superdiagonal.

Examples

source

`LinearAlgebra.tril!` – Function.

```
tril!(M)
```

Lower triangle of a matrix, overwriting M in the process. See also `tril`.

source

```
tril!(M, k::Integer)
```

Return the lower triangle of M starting from the kth superdiagonal, overwriting M in the process.

Examples

source

LinearAlgebra.diagind – Function.

```
diagind(M, k::Integer=0)
```

An `AbstractRange` giving the indices of the kth diagonal of the matrix M.

Examples

source

LinearAlgebra.diag – Function.

```
diag(M, k::Integer=0)
```

The kth diagonal of a matrix, as a vector.

See also: diagm

Examples

source

LinearAlgebra.diagm – Function.

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a square matrix from `Pairs` of diagonals and vectors. Vector `kv.second` will be placed on the `kv.first` diagonal. `diagm` constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see Diagonal, Bidiagonal Tridiagonal and SymTridiagonal.

Examples

source

LinearAlgebra.rank – Function.

```
rank(A[, tol::Real])
```

Compute the rank of a matrix by counting how many singular values of A have magnitude greater than `tol*σ` where σ is A's largest singular values. By default, the value of `tol` is the smallest dimension of A multiplied by the eps of the eltype of A.

Examples

source

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

source

```
norm(x::Number, p::Real=2)
```

For numbers, return $(|x|^p)^{1/p}$.

Examples

source

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:

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$

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:

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$$

Examples

source

```
opnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to norm.

source

```
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

source

LinearAlgebra.normalize! – Function.

```
normalize!(v::AbstractVector, p::Real=2)
```

Normalize the vector v in-place so that its p-norm equals unity, i.e. norm(v, p) == 1. See also normalize and norm.

source

LinearAlgebra.normalize – Function.

```
normalize(v::AbstractVector, p::Real=2)
```

Normalize the vector v so that its p-norm equals unity, i.e. norm(v, p) == vecnorm(v, p) == 1. See also normalize! and norm.

Examples

source

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.

source

LinearAlgebra.condskeel – Function.

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M, p) = \left\||M|\,|M^{-1}|\right\|_p$$
$$\kappa_S(M, x, p) = \left\||M|\,|M^{-1}|\,|x|\right\|_p$$

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.

source

`LinearAlgebra.tr` – Function.

> `tr(M)`

Matrix trace. Sums the diagonal elements of `M`.

Examples

source

`LinearAlgebra.det` – Function.

> `det(M)`

Matrix determinant.

Examples

source

`LinearAlgebra.logdet` – Function.

> `logdet(M)`

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

source

`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

source

`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

source

`LinearAlgebra.pinv` – Function.

> `pinv(M[, tol::Real])`

Computes the Moore-Penrose pseudoinverse.

For matrices `M` with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values above a given threshold, `tol`.

The optimal choice of `tol` varies both with the value of `M` and the intended application of the pseudoinverse. The default value of `tol` is `eps(real(float(one(eltype(M)))))*minimum(size(M))`, which is essentially machine epsilon for the real part of a matrix element multiplied by the larger matrix dimension. For inverting dense ill-conditioned matrices in a least-squares sense, `tol = sqrt(eps(real(float(one(eltype(M))))))` is recommended.

For more information, see [4], [5], [6], [7].

Examples

source

`LinearAlgebra.nullspace` – Function.

> `nullspace(M[, tol::Real])`

Computes a basis for the nullspace of `M` by including the singular vectors of A whose singular have magnitude are greater than `tol*σ`, where σ is A's largest singular values. By default, the value of `tol` is the smallest dimension of A multiplied by the eps of the eltype of A.

Examples

source

`Base.kron` – Function.

> `kron(A, B)`

Kronecker tensor product of two vectors or two matrices.

Examples

source

`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

source

---

[4] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[5] Åke Björck, "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

[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.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

> `source`

`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 symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the square root. Otherwise, the square root is determined by means of the Björck-Hammarling method [11], which computes the complex Schur form (`schur`) and then the complex square root of the triangular factor.

Examples

> `source`

`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

> `source`

`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`.

---

[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 Fréchet 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] Åke Björck 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

Examples

source

Base.Math.sincos – Method.

```
sincos(A::AbstractMatrix)
```

Compute the matrix sine and cosine of a square matrix A.

Examples

source

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

source

Base.Math.sec – Method.

```
sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix A.

source

Base.Math.csc – Method.

```
csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix A.

source

Base.Math.cot – Method.

```
cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix A.

source

Base.cosh – Method.

```
cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix A.

source

Base.sinh – Method.

```
sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix A.

source

Base.tanh – Method.

```
tanh(A::AbstractMatrix)
```

Compute the matrix hyperbolic tangent of a square matrix A.

source

Base.Math.sech – Method.

```
sech(A::AbstractMatrix)
```

Compute the matrix hyperbolic secant of square matrix A.

source

Base.Math.csch – Method.

```
csch(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosecant of square matrix A.

source

Base.Math.coth – Method.

```
coth(A::AbstractMatrix)
```

Compute the matrix hyperbolic cotangent of square matrix A.

source

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

source

Base.asin – Method.

```
asin(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

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

`source`

`Base.atan` – Method.

```
atan(A::AbstractMatrix)
```

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

`source`

`Base.Math.asec` – Method.

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of A.

`source`

`Base.Math.acsc` – Method.

```
acsc(A::AbstractMatrix)
```

Compute the inverse matrix cosecant of A.

`source`

`Base.Math.acot` – Method.

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of A.

`source`

`Base.acosh` – Method.

```
acosh(A::AbstractMatrix)
```

---

[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

[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

Compute the inverse hyperbolic matrix cosine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [15].

source

`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].

source

`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].

source

`Base.Math.asech` – Method.

```
asech(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic secant of A.

source

`Base.Math.acsch` – Method.

```
acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of A.

source

`Base.Math.acoth` – Method.

```
acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of A.

source

`LinearAlgebra.lyap` – Function.

```
lyap(A, C)
```

---

[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

[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

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

source

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

source

LinearAlgebra.issuccess – Function.

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

source

LinearAlgebra.issymmetric – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

source

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

source

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

source

LinearAlgebra.istril – Function.

```
istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is lower triangular starting from the kth superdiagonal.

Examples

source

`LinearAlgebra.istriu` – Function.

```
istriu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is upper triangular starting from the kth superdiagonal.

Examples

source

`LinearAlgebra.isdiag` – Function.

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal.

Examples

source

`LinearAlgebra.ishermitian` – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

source

`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

source

`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

source

`Base.adjoint` – Function.

```
adjoint(A)
```

Lazy adjoint (conjugate transposition) (also postfix `'`). Note that `adjoint` is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

source

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

source

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

source

LinearAlgebra.stride1 – Function.

```
stride1(A) -> Int
```

Return the distance between successive array elements in dimension 1 in units of element size.

Examples

source

LinearAlgebra.checksquare – Function.

```
LinearAlgebra.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

source

71.4   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

source

`LinearAlgebra.lmul!` – Function.

```
lmul!(a::Number, B::AbstractArray)
```

Scale an array B by a scalar a overwriting B in-place.

Examples

source

```
lmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting B, and return the result.

Examples

source

`LinearAlgebra.rmul!` – Function.

```
rmul!(A::AbstractArray, b::Number)
```

Scale an array A by a scalar b overwriting A in-place.

Examples

source

```
rmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting A, and return the result.

Examples

source

`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

`source`

```
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

`source`

`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.

`source`

71.5 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.

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.

Triangle Referencing

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

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.

LinearAlgebra.BLAS – Module.

Interface to BLAS subroutines.

source

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

source

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

source

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.

source

LinearAlgebra.BLAS.nrm2 – Function.

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of n elements of array X with stride `incx`.

Examples

source

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

source

LinearAlgebra.axpy! – Function.

```
axpy!(a, X, Y)
```

Overwrite Y with a*X + Y, where a is a scalar. Return Y.

Examples

source

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.

source

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`.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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 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.

source

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.

source

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.

source

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.

source

LinearAlgebra.BLAS.gemm – Method.

```
gemm(tA, tB, alpha, A, B)
```

Return `alpha*A*B` or the other three variants according to `tA` and tB.

source

LinearAlgebra.BLAS.gemm – Method.

```
gemm(tA, tB, A, B)
```

Return A*B or the other three variants according to `tA` and tB.

source

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.

source

LinearAlgebra.BLAS.gemv – Method.

> gemv(tA, alpha, A, x)

Return `alpha*A*x` or `alpha*A'x` according to tA. `alpha` is a scalar.

> source

LinearAlgebra.BLAS.gemv – Method.

> gemv(tA, A, x)

Return `A*x` or `A'x` according to tA.

> source

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.

> source

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.

> source

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.

> source

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.

> source

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.

> source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

`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.

source

`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.

source

`LinearAlgebra.BLAS.set_num_threads` – Function.

```
set_num_threads(n)
```

Set the number of threads the BLAS library should use.

source

`LinearAlgebra.I` – Constant.

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Examples

source

71.6   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.

source

`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.

source

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.

source

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.

source

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).

source

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.

source

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.

source

```
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.

source

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.

source

```
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.

source

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.

source

```
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.

source

LinearAlgebra.LAPACK.geqp3! – Function.

```
geqp3!(A, jpvt, tau)
```

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. jpvt must have length length greater than or equal to n if A is an (m x n) matrix. tau must have length greater than or equal to the smallest dimension of A.

A, jpvt, and tau are modified in-place.

source

```
geqp3!(A, jpvt) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. jpvt must have length greater than or equal to n if A is an (m x n) matrix.

Returns A and jpvt, modified in-place, and tau, which stores the elementary reflectors.

source

```
geqp3!(A) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3.

Returns A, modified in-place, `jpvt`, which represents the pivoting matrix P, and `tau`, which stores the elementary reflectors.

source

`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.

source

```
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.

source

`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.

source

```
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.

source

`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.

source

```
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.

source

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`).

source

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.

source

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.

source

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.

source

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.

source

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).

source

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.

source

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.

source

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of gesvx!.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

[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.

[source](#)

[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.

[source](#)

[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.

[source](#)

[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.

[source](#)

[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`.

[source](#)

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

> source

`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.

> source

`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.

> source

`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.

> source

`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.

> source

`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.

> source

`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.

    source

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.

    source

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.

    source

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.

    source

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.

    source

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.

    source

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.

source

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.

source

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.

source

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.

source

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.

source

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`.

source

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.

source

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.

source

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.

source

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`.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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.

source

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 = 0` 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.

source

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.

source

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!`.

source

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 over-written by its Schur form.

Returns A, vs containing the Schur vectors, and w, containing the eigenvalues.

source

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.

source

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.

source

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.

source

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.

source

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`.

source

Chapter 72

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:

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

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

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`.

72.1 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 `AbstractStrings` 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 unique for each logging macro invocation. This is very useful as a key for caching information or actions associated with an event. For instance, it can be used to limit the number of times a message is presented to the user.

- 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.

72.2 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`.

72.3 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.

72.4 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`.

72.5 Reference

Creating events

`Base.CoreLogging.@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

`Base.CoreLogging.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

`Base.CoreLogging.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

`Base.CoreLogging.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 Symbol to be used as a key to identify the log statement when filtering.

> source

`Base.CoreLogging.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

`Base.CoreLogging.min_enabled_level` – Function.

```
min_enabled_level(logger)
```

Return the maximum disabled level for `logger` for early filtering. That is, the log level below or equal to which all messages are filtered.

> source

`Base.CoreLogging.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

`Base.CoreLogging.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:

`Base.CoreLogging.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

`Base.CoreLogging.with_logger` – Function.

```
with_logger(function, logger)
```

Execute `function`, directing all log messages to `logger`.

Example

source

`Base.CoreLogging.current_logger` – Function.

```
current_logger()
```

Return the logger for the current task, or the global logger if none is is attached to the task.

source

Loggers that are supplied with the system:

`Base.CoreLogging.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).

source

Base.CoreLogging.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 73

Markdown

Chapter 74

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

source

`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

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).

source

```
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

This creates a 25-by-30000 `BitArray`, linked to the file associated with stream `io`.

source

`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.

source

Chapter 75

Pkg

75.1 Introduction

Pkg is the standard package manager for Julia 1.0 and newer. Unlike traditional package managers, which install and manage a single global set of packages, Pkg is designed around "environments": independent sets of packages that can be local to an individual project or shared and selected by name. The exact set of packages and versions in an environment is captured in a *manifest file* which can be checked into a project repository and tracked in version control, significantly improving reproducibility of projects. If you've ever tried to run code you haven't used in a while only to find that you can't get anything to work because you've updated or uninstalled some of the packages your project was using, you'll understand the motivation for this approach. In Pkg, since each project maintains its own independent set of package versions, you'll never have this problem again. Moreover, if you check out a project on a new system, you can simply materialize the environment described by its manifest file and immediately be up and running with a known-good set of dependencies.

Since environments are managed and updated independently from each other, "dependency hell" is significantly alleviated in Pkg. If you want to use the latest and greatest version of some package in a new project but you're stuck on an older version in a different project, that's no problem – since they have separate environments they can just use different versions, which are both installed at the same time in different locations on your system. The location of each package version is canonical, so when environments use the same versions of packages, they can share installations, avoiding unnecessary duplication of the package. Old package versions that are no longer used by any environments are periodically "garbage collected" by the package manager.

Pkg's approach to local environments may be familiar to people who have used Python's `virtualenv` or Ruby's `bundler`. In Julia, instead of hacking the language's code loading mechanisms to support environments, we have the benefit that Julia natively understands them. In addition, Julia environments are "stackable": you can overlay one environment with another and thereby have access to additional packages outside of the primary environment. This makes it easy to work on a project, which provides the primary environment, while still having access to all your usual dev tools like profilers, debuggers, and so on, just by having an environment including these dev tools later in the load path.

Last but not least, Pkg is designed to support federated package registries. This means that it allows multiple registries managed by different parties to interact seamlessly. In particular, this includes private registries which can live behind corporate firewalls. You can install and update your own packages from a private registry with exactly the same tools and workflows that you use to install and manage official Julia packages. If you urgently need to apply a hotfix for a public package that's critical to your company's product, you can tag a private version of it in your company's internal registry and get a fix to your developers and ops teams quickly and easily without having to wait for an upstream patch to be accepted and published. Once an official fix is published, however, you can just upgrade your dependencies and you'll be back on an official release again.

75.2 Glossary

Project: a source tree with a standard layout, including a `src` directory for the main body of Julia code, a `test` directory for testing the project, `docs` for documentation files, and optionally a `deps` directory for a build script and its outputs. A project will typically also have a project file and may optionally have a manifest file:

- **Project file:** a file in the root directory of a project, named `Project.toml` (or `JuliaProject.toml`) describing metadata about the project, including its name, UUID (for packages), authors, license, and the names and UUIDs of packages and libraries that it depends on.

- **Manifest file:** a file in the root directory of a project, named `Manifest.toml` (or `JuliaManifest.toml`) describing a complete dependency graph and exact versions of each package and library used by a project.

Package: a project which provides reusable functionality that can be used by other Julia projects via `import X` or `using X`. A package should have a project file with a `uuid` entry giving its package UUID. This UUID is used to identify the package in projects that depend on it.

> **Note**
>
> For legacy reasons it is possible to load a package without a project file or UUID from the REPL or the top-level of a script. It is not possible, however, to load a package without a project file or UUID from a project with them. Once you've loaded from a project file, everything needs a project file and UUID.

Application: a project which provides standalone functionality not intended to be reused by other Julia projects. For example a web application or a commmand-line utility, or simulation/analytics code accompanying a scientific paper. An application may have a UUID but does not need one. An application may also provide global configuration options for packages it depends on. Packages, on the other hand, may not provide global configuration since that could conflict with the configuration of the main application.

> **Note**
>
> **Projects *vs.* Packages *vs.* Applications:**
>
> 1. **Project** is an umbrella term: packages and applications are kinds of projects.
> 2. **Packages** should have UUIDs, applications can have a UUIDs but don't need them.
> 3. **Applications** can provide global configuration, whereas packages cannot.

Library (future work): a compiled binary dependency (not written in Julia) packaged to be used by a Julia project. These are currently typically built in- place by a `deps/build.jl` script in a project's source tree, but in the future we plan to make libraries first-class entities directly installed and upgraded by the package manager.

Environment: the combination of the top-level name map provided by a project file combined with the dependency graph and map from packages to their entry points provided by a manifest file. For more detail see the manual section on code loading.

- **Explicit environment:** an environment in the form of an explicit project file and an optional corresponding manifest file together in a directory. If the manifest file is absent then the implied dependency graph and location maps are empty.

- **Implicit environment:** an environment provided as a directory (without a project file or manifest file) containing packages with entry points of the form `X.jl`, `X.jl/src/X.jl` or `X/src/X.jl`. The top-level name map is implied by these entry points. The dependency graph is implied by the existence of project files inside of these package directories, e.g. `X.jl/Project.toml` or `X/Project.toml`. The dependencies of the X package are the dependencies in the corresponding project file if there is one. The location map is implied by the entry points themselves.

Registry: a source tree with a standard layout recording metadata about a registered set of packages, the tagged versions of them which are available, and which versions of packages are compatible or incompatible with each other. A registry is indexed by package name and UUID, and has a directory for each registered package providing the following metadata about it:

- name – e.g. `DataFrames`

- UUID – e.g. `a93c6f00-e57d-5684-b7b6-d8193f3e46c0`

- authors – e.g. `Jane Q. Developer <jane@example.com>`

- license – e.g. MIT, BSD3, or GPLv2

- repository – e.g. `https://github.com/JuliaData/DataFrames.jl.git`

- description – a block of text summarizing the functionality of a package

- keywords – e.g. `data`, `tabular`, `analysis`, `statistics`

- versions – a list of all registered version tags

For each registered version of a package, the following information is provided:

- its semantic version number – e.g. `v1.2.3`

- its git tree SHA-1 hash – e.g. `7ffb18ea3245ef98e368b02b81e8a86543a11103`

- a map from names to UUIDs of dependencies

- which versions of other packages it is compatible/incompatible with

Dependencies and compatibility are stored in a compressed but human-readable format using ranges of package versions.

Depot: a directory on a system where various package-related resources live, including:

- `environments`: shared named environments (e.g. `v0.7`, `devtools`)

- `clones`: bare clones of package repositories

- `compiled`: cached compiled package images (`.ji` files)

- `config`: global configuration files (e.g. `startup.jl`)

- `dev`: default directory for package development

- `logs`: log files (e.g. `manifest_usage.toml`, `repl_history.jl`)

- `packages`: installed package versions

- `registries`: clones of registries (e.g. `General`)

Load path: a stack of environments where package identities, their dependencies, and entry-points are searched for. The load path is controlled in Julia by the `LOAD_PATH` global variable which is populated at startup based on the value of the `JULIA_LOAD_PATH` environment variable. The first entry is your primary environment, often the current project, while later entries provide additional packages one may want to use from the REPL or top-level scripts.

Depot path: a stack of depot locations where the package manager, as well as Julia's code loading mechanisms, look for registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. The depot path is controlled by the Julia `DEPOT_PATH` global variable which is populated at startup based on the value of the `JULIA_DEPOT_PATH` environment variable. 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.

75.3 Getting Started

The Pkg REPL-mode is entered from the Julia REPL using the key `]`.

```
(v0.7) pkg>
```

The part inside the parenthesis of the prompt shows the name of the current project. Since we haven't created our own project yet, we are in the default project, located at `~/.julia/environments/v0.7` (or whatever version of Julia you happen to run).

To return to the `julia>` prompt, either press backspace when the input line is empty or press Ctrl+C. Help is available by calling `pkg> help`. If you are in an environment that does not have access to a REPL you can still use the REPL mode commands using the string macro `pkg` available after `using Pkg`. The command `pkg"cmd"` would be equivalent to executing `cmd` in the REPL mode.

The documentation here describes using Pkg from the REPL mode. Documentation of using the Pkg API (by calling `Pkg.` functions) is in progress of being written.

Adding packages

There are two ways of adding packages, either using the `add` command or the `dev` command. The most frequently used one is add and its usage is described first.

Adding registered packages

In the Pkg REPL packages can be added with the `add` command followed by the name of the package, for example:

```
(v0.7) pkg> add Example
   Cloning default registries into /Users/kristoffer/.julia/registries
   Cloning registry General from "https://github.com/JuliaRegistries/General.git"
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test
```

Here we added the package Example to the current project. In this example, we are using a fresh Julia installation, and this is our first time adding a package using Pkg. By default, Pkg clones Julia's General registry, and uses this registry to look up packages requested for inclusion in the current environment. The status update shows a short form of the package UUID to the left, then the package name, and the version. Since standard libraries (e.g. `Test`) are shipped with Julia, they do not have a version. The project status contains the packages you have added yourself, in this case, `Example`:

```
(v0.7) pkg> st
    Status `Project.toml`
  [7876af07] Example v0.5.1
```

The manifest status, in addition, includes the dependencies of explicitly added packages.

```
(v0.7) pkg> st --manifest
    Status `Manifest.toml`
  [7876af07] Example v0.5.1
  [8dfed614] Test
```

It is possible to add multiple packages in one command as pkg> `add A B C`.

After a package is added to the project, it can be loaded in Julia:

```
julia> using Example

julia> Example.hello("User")
"Hello, User"
```

A specific version can be installed by appending a version after a @ symbol, e.g. `@v0.4`, to the package name:

```
(v0.7) pkg> add Example@0.4
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.4.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.4.1
```

If the master branch (or a certain commit SHA) of `Example` has a hotfix that has not yet included in a registered version, we can explicitly track a branch (or commit) by appending #branch (or #commit) to the package name:

```
(v0.7) pkg> add Example#master
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
```

The status output now shows that we are tracking the `master` branch of `Example`. When updating packages, we will pull updates from that branch.

To go back to tracking the registry version of `Example`, the command `free` is used:

```
(v0.7) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)  v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1+ #master )https://github.com/JuliaLang/Example.jl.git)  v0.5.1
```

Adding unregistered packages

If a package is not in a registry, it can still be added by instead of the package name giving the URL to the repository to add.

```
(v0.7) pkg> add https://github.com/fredrikekre/ImportMacros.jl
  Updating git-repo `https://github.com/fredrikekre/ImportMacros.jl`
 Resolving package versions...
Downloaded MacroTools  v0.4.1
  Updating `~/.julia/environments/v0.7/Project.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  [1914dd2f] + MacroTools v0.4.1
```

The dependencies of the unregistered package (here `MacroTools`) got installed. For unregistered packages we could have given a branch (or commit SHA) to track using #, just like for registered packages.

Adding a local package

Instead of giving a URL of a git repo to add we could instead have given a local path to a git repo. This works similarly to adding a URL. The local repository will be tracked (at some branch) and updates from that local repo are pulled when packages are updated. Note that changes to files in the local package repository will not immediately be reflected when loading that package. The changes would have to be committed and the packages updated in order to pull in the changes.

Developing packages

By only using add your Manifest will always have a "reproducible state", in other words, as long as the repositories and registries used are still accessible it is possible to retrieve the exact state of all the dependencies in the project. This has the advantage that you can send your project (`Project.toml` and `Manifest.toml`) to someone else and they can "instantiate" that project in the same state as you had it locally. However, when you are developing a package, it is more convenient to load packages at their current state at some path. For this reason, the dev command exists.

Let's try to dev a registered package:

```
(v0.7) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
```

The dev command fetches a full clone of the package to ~/.julia/dev/ (the path can be changed by setting the environment variable JULIA_PKG_DEVDIR). When importing `Example` julia will now import it from ~/.julia/dev/Example and whatever local changes have been made to the files in that path are consequently reflected in the code loaded. When we used add we said that we tracked the package repository, we here say that we track the path itself. Note that the package manager will never touch any of the files at a tracked path. It is therefore up to you to pull updates, change branches etc. If we try to dev a package at some branch that already exists at ~/.julia/dev/ the package manager we will simply use the existing path. For example:

```
(v0.7) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
[ Info: Path `/Users/kristoffer/.julia/dev/Example` exists and looks like the correct package,
    using existing path instead of cloning
```

Note the info message saying that it is using the existing path. As a general rule, the package manager will never touch files that are tracking a path.

If dev is used on a local path, that path to that package is recorded and used when loading that package. The path will be recorded relative to the project file, unless it is given as an absolute path.

To stop tracking a path and use the registered version again, use `free`

```
(v0.7) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07]  Example v0.5.1+ [`~/.julia/dev/Example`]  v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07]  Example v0.5.1+ [`~/.julia/dev/Example`]  v0.5.1
```

It should be pointed out that by using dev your project is now inherently stateful. Its state depends on the current content of the files at the path and the manifest cannot be "instantiated" by someone else without knowing the exact content of all the packages that are tracking a path.

Note that if you add a dependency to a package that tracks a local path, the Manifest (which contains the whole dependency graph) will become out of sync with the actual dependency graph. This means that the package will not be able to load that dependency since it is not recorded in the Manifest. To update sync the Manifest, use the REPL command `resolve`.

Removing packages

Packages can be removed from the current project by using `pkg> rm Package`. This will only remove packages that exist in the project, to remove a package that only exists as a dependency use `pkg> rm --manifest DepPackage`. Note that this will remove all packages that depends on `DepPackage`.

Updating packages

When new versions of packages the project is using are released, it is a good idea to update. Simply calling up will try to update *all* the dependencies of the project to the latest compatible version. Sometimes this is not what you want. You can specify a subset of the dependencies to upgrade by giving them as arguments to up, e.g:

```
(v0.7) pkg> up Example
```

The version of all other packages direct dependencies will stay the same. If you only want to update the minor version of packages, to reduce the risk that your project breaks, you can give the `--minor` flag, e.g:

```
(v0.7) pkg> up --minor Example
```

Packages that track a repository are not updated when a minor upgrade is done. Packages that track a path are never touched by the package manager.

Pinning a package

A pinned package will never be updated. A package can be pinned using `pin` as for example

```
(v0.7) pkg> pin Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1  v0.5.1
```

Note the pin symbol showing that the package is pinned. Removing the pin is done using `free`

```
(v0.7) pkg> free Example
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1   v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1   v0.5.1
```

Testing packages

The tests for a package can be run using `test`command:

```
(v0.7) pkg> test Example
   Testing Example
   Testing Example tests passed
```

Building packages

The build step of a package is automatically run when a package is first installed. The output of the build process is directed to a file. To explicitly run the build step for a package the `build` command is used:

```
(v0.7) pkg> build MbedTLS
  Building MbedTLS  `~/.julia/packages/MbedTLS/h1Vu/deps/build.log`

shell> cat ~/.julia/packages/MbedTLS/h1Vu/deps/build.log
 Warning: `wait(t::Task)` is deprecated, use `fetch(t)` instead.
   caller = macro expansion at OutputCollector.jl:63 [inlined]
 @ Core OutputCollector.jl:63
...
[ Info: using prebuilt binaries
```

75.4   Creating your own projects

So far we have added packages to the default project at ~/.julia/environments/v0.7, it is, however, easy to create other, independent, projects. It should be pointed out if two projects uses the same package at the same version, the content of this package is not duplicated. In order to create a new project, create a directory for it and then activate that directory to make it the "active project" which package operations manipulate:

```
shell> mkdir MyProject

shell> cd MyProject
/Users/kristoffer/MyProject

(v0.7) pkg> activate .

(MyProject) pkg> st
    Status `Project.toml`
```

Note that the REPL prompt changed when the new project is activated. Since this is a newly created project, the status command show it contains no packages, and in fact, it has no project or manifest file until we add a package to it:

```
shell> ls -l
total 0

(MyProject) pkg> add Example
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
```

```
 Resolving package versions...
  Updating `Project.toml`
  [7876af07] + Example v0.5.1
  Updating `Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test

shell> ls -l
total 8
-rw-r--r-- 1 stefan staff 207 Jul  3 16:35 Manifest.toml
-rw-r--r-- 1 stefan staff  56 Jul  3 16:35 Project.toml

shell> cat Project.toml
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"

shell> cat Manifest.toml
[[Example]]
deps = ["Test"]
git-tree-sha1 = "8eb7b4d4ca487caade9ba3e85932e28ce6d6e1f8"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "0.5.1"

[[Test]]
uuid = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
```

This new environment is completely separate from the one we used earlier.

75.5 Garbage collecting old, unused packages

As packages are updated and projects are deleted, installed packages that were once used will inevitably become old and not used from any existing project. Pkg keeps a log of all projects used so it can go through the log and see exactly which projects still exist and what packages those projects used. The rest can be deleted. This is done with the gc command:

```
(v0.7) pkg> gc
    Active manifests at:
        `/Users/kristoffer/BinaryProvider/Manifest.toml`
        ...
        `/Users/kristoffer/Compat.jl/Manifest.toml`
  Deleted /Users/kristoffer/.julia/packages/BenchmarkTools/1cAj: 146.302 KiB
  Deleted /Users/kristoffer/.julia/packages/Cassette/BXVB: 795.557 KiB
  ...
  Deleted /Users/kristoffer/.julia/packages/WeakRefStrings/YrK6: 27.328 KiB
  Deleted 36 package installations: 113.205 MiB
```

Note that only packages in ~/.julia/packages are deleted.

75.6 Creating your own packages

A package is a project with a name, uuid and version entry in the Project.toml file src/PackageName.jl file that defines the module PackageName. This file is executed when the package is loaded.

Generating files for a package

To generate files for a new package, use `pkg> generate`.

```
(v0.7) pkg> generate HelloWorld
```

This creates a new project `HelloWorld` with the following files (visualized with the external `tree command`):

```
shell> cd HelloWorld

shell> tree .
.
 Project.toml
 src
     HelloWorld.jl

1 directory, 2 files
```

The `Project.toml` file contains the name of the package, its unique UUID, its version, the author and eventual dependencies:

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
author = ["Some One <someone@email.com>"]

[deps]
```

The content of `src/HelloWorld.jl` is:

```
module HelloWorld

greet() = print("Hello World!")

end # module
```

We can now activate the project and load the package:

```
pkg> activate .

julia> import HelloWorld

julia> HelloWorld.greet()
Hello World!
```

Adding dependencies to the project

Let's say we want to use the standard library package Random and the registered package JSON in our project. We simply add these packages (note how the prompt now shows the name of the newly generated project, since we are inside the `HelloWorld` project directory):

```
(HelloWorld) pkg> add Random JSON
 Resolving package versions...
  Updating "~/Documents/HelloWorld/Project.toml"
 [682c06a0] + JSON v0.17.1
 [9a3f8284] + Random
  Updating "~/Documents/HelloWorld/Manifest.toml"
 [34da2185] + Compat v0.57.0
```

```
  [682c06a0] + JSON v0.17.1
  [4d1e1d77] + Nullables v0.0.4
  ...
```

Both Random and JSON got added to the project's `Project.toml` file, and the resulting dependencies got added to the `Manifest.toml` file. The resolver has installed each package with the highest possible version, while still respecting the compatibility that each package enforce on its dependencies.

We can now use both Random and JSON in our project. Changing `src/HelloWorld.jl` to

```
module HelloWorld

import Random
import JSON

greet() = print("Hello World!")
greet_alien() = print("Hello ", Random.randstring(8))

end # module
```

and reloading the package, the new `greet_alien` function that uses Random can be used:

```
julia> HelloWorld.greet_alien()
Hello aT157rHV
```

Adding a build step to the package.

The build step is executed the first time a package is installed or when explicitly invoked with `build`. A package is built by executing the file `deps/build.jl`.

```
shell> cat deps/build.log
I am being built...

(HelloWorld) pkg> build
  Building HelloWorld `deps/build.log`
 Resolving package versions...

shell> cat deps/build.log
I am being built...
```

If the build step fails, the output of the build step is printed to the console

```
shell> cat deps/build.jl
error("Ooops")

(HelloWorld) pkg> build
  Building HelloWorld `deps/build.log`
 Resolving package versions...
 Error: Error building `HelloWorld`:
 ERROR: LoadError: Ooops
 Stacktrace:
  [1] error(::String) at ./error.jl:33
  [2] top-level scope at none:0
  [3] include at ./boot.jl:317 [inlined]
  [4] include_relative(::Module, ::String) at ./loading.jl:1071
  [5] include(::Module, ::String) at ./sysimg.jl:29
  [6] include(::String) at ./client.jl:393
```

```
   [7] top-level scope at none:0
 in expression starting at /Users/kristoffer/.julia/dev/Pkg/HelloWorld/deps/build.jl:1
 @ Pkg.Operations Operations.jl:938
```

Adding tests to the package

When a package is tested the file `test/runtests.jl` is executed.

```
shell> cat test/runtests.jl
println("Testing...")
(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
Testing...
   Testing HelloWorld tests passed
```

Test-specific dependencies

Sometimes one might want to use some packages only at testing time but not enforce a dependency on them when the package is used. This is possible by adding dependencies to `[extras]` and a `test` target in `[targets]` to the Project file. Here we add the `Test` standard library as a test-only dependency by adding the following to the Project file:

```
[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Test"]
```

We can now use `Test` in the test script and we can see that it gets installed on testing:

```
shell> cat test/runtests.jl
using Test
@test 1 == 1

(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Project.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
  [8dfed614] + Test
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Manifest.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
   Testing HelloWorld tests passed```
```

Compatibility

Compatibility refers to the ability to restrict what version of the dependencies that your project is compatible with. If the compatibility for a dependency is not given, the project is assumed to be compatible with all versions of that dependency.

Compatibility for a dependency is entered in the `Project.toml` file as for example:

```
[compat]
Example = "0.4.3"
```

After a compatibility entry is put into the project file, up can be used to apply it.

The format of the version specifier is described in detail below.

> **Info**
>
> There is currently no way to give compatibility from the Pkg REPL mode so for now, one has to manually edit the project file.

Version specifier format

Similar to other package managers, the Julia package manager respects semantic versioning (semver). As an example, a version specifier is given as e.g. `1.2.3` is therefore assumed to be compatible with the versions `[1.2.3 - 2.0.0)` where `)` is a non-inclusive upper bound. More specifically, a version specifier is either given as a **caret specifier**, e.g. `^1.2.3` or a **tilde specifier** `~1.2.3`. Caret specifiers are the default and hence `1.2.3 == ^1.2.3`. The difference between a caret and tilde is described in the next section. The intersection of multiple version specifiers can be formed by comma separating indiviual version specifiers.

Caret specifiers   A caret specifier allows upgrade that would be compatible according to semver. An updated dependency is considered compatible if the new version does not modify the left-most non zero digit in the version specifier.

Some examples are shown below.

```
^1.2.3 = [1.2.3, 2.0.0)
^1.2 = [1.2.0, 2.0.0)
^1 =  [1.0.0, 2.0.0)
^0.2.3 = [0.2.3, 0.3.0)
^0.0.3 = [0.0.3, 0.0.4)
^0.0 = [0.0.0, 0.1.0)
^0 = [0.0.0, 1.0.0)
```

While the semver specification says that all versions with a major version of 0 are incompatible with each other, we have made that choice that a version given as `0.a.b` is considered compatible with `0.a.c` if `a != 0` and `c >= b`.

Tilde specifiers   A tilde specifier provides more limited upgrade possibilities. With a tilde, only the last specified digit is allowed to increment by one. This gives the following example.

```
~1.2.3 = [1.2.3, 1.2.4)
~1.2 = [1.2.0, 1.3.0)
~1 = [1.0.0, 2.0.0)
```

Inequality specifiers

Inequalities can also be used to specify version ranges:

```
>= 1.2.3 = [1.2.3,  ∞)≥
  1.2.3 = [1.2.3,  ∞)
= 1.2.3 = [1.2.3, 1.2.3]
< 1.2.3 = [0.0.0, 1.2.2]
```

75.7   Precompiling a project

The REPL command `precompile` can be used to precompile all the dependencies in the project. You can for example do

```
(HelloWorld) pkg> update; precompile
```

to update the dependencies and then precompile them.

75.8   Preview mode

If you just want to see the effects of running a command, but not change your state you can `preview` a command. For example:

```
(HelloWorld) pkg> preview add Plots
```

or

```
(HelloWorld) pkg> preview up
```

will show you the effects of adding `Plots`, or doing a full upgrade, respectively, would have on your project. However, nothing would be installed and your `Project.toml` and `Manifest.toml` are untouched.

75.9   Using someone else's project

Simply clone their project using e.g. `git clone`, cd to the project directory and call

```
(v0.7) pkg> activate .

(SomeProject) pkg> instantiate
```

If the project contains a manifest, this will install the packages in the same state that is given by that manifest. Otherwise, it will resolve the latest versions of the dependencies compatible with the project.

75.10   References

This section describes the "API mode" of interacting with Pkg.jl which is recommended for non-interactive usage, in i.e. scripts. In the REPL mode packages (with associated version, UUID, URL etc) are parsed from strings, for example, `"Package#master"`,`"Package@v0.1"`, `"www.mypkg.com/MyPkg#my/feature"`. It is possible to use strings as arguments for simple commands in the API mode (like `Pkg.add(["PackageA",  "PackageB"])`, more complicated commands, that e.g. specify URLs or version range, uses a more structured format over strings. This is done by creating an instance of a PackageSpec which are passed in to functions.

Pkg.PackageSpec – Type.

```
PackageSpec(name::String, [uuid::UUID, version::VersionNumber])
PackageSpec(; name, url, path, rev, version, mode, level)
```

A PackageSpec is a representation of a package with various metadata. This includes:

- The name of the package.

- The package unique uuid.

- A version (for example when adding a package. When upgrading, can also be an instance of

the enum UpgradeLevel

- A url and an optional git revision. rev could be a branch name or a git commit SHA.

- A local path path. This is equivalent to using the url argument but can be more descriptive.

- A mode, which is an instance of the enum `PackageMode` which can be either PKGMODE_PROJECT or

PKGMODE_MANIFEST, defaults to PKGMODE_PROJECT. Used in e.g. `Pkg.rm`.

Most functions in Pkg take a `Vector` of PackageSpec and do the operation on all the packages in the vector.

Below is a comparison between the REPL version and the PackageSpec version:

REPL	API
Package	PackageSpec("Package")
Package@0.2	PackageSpec(name="Package", version="0.2")
Package=a67d...	PackageSpec(name="Package", uuid="a67d..."
Package#master	PackageSpec(name="Package", rev="master")
local/path#feature	PackageSpec(path="local/path"; rev="feature)
www.mypkg.com	PackageSpec(url="www.mypkg.com")
--manifest Package	PackageSpec(name="Package", mode=PKGSPEC_MANIFEST)
--major Package	PackageSpec(name="Package", version=PKGLEVEL_MAJOR)

source

`Pkg.PackageMode` – Type.

> `PackageMode`

An enum with the instances

- `PKGMODE_MANIFEST`
- `PKGMODE_PROJECT`

Determines if operations should be made on a project or manifest level. Used as an argument to `PackageSpec` or as an argument to `Pkg.rm`.

source

`Pkg.UpgradeLevel` – Type.

> `UpgradeLevel`

An enum with the instances

- `UPLEVEL_FIXED`
- `UPLEVEL_PATCH`
- `UPLEVEL_MINOR`
- `UPLEVEL_MAJOR`

Determines how much a package is allowed to be updated. Used as an argument to `PackageSpec` or as an argument to `Pkg.update`.

source

`Pkg.add` – Function.

```
Pkg.add(pkg::Union{String, Vector{String})
Pkg.add(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Add a package to the current project. This package will be available using the `import` and `using` keywords in the Julia REPL and if the current project is a package, also inside that package.

Examples

See also PackageSpec.

source

Pkg.develop – Function.

```
Pkg.develop(pkg::Union{String, Vector{String})
Pkg.develop(pkgs::Union{Packagespec, Vector{Packagespec}})
```

Make a package available for development by tracking it by path. If `pkg` is given with only a name or by a URL the packages will be downloaded to the location by the environment variable JULIA_PKG_DEVDIR with `.julia/dev` as the default.

If `pkg` is given as a local path, the package at that path will be tracked.

Examples

See also PackageSpec

source

Pkg.activate – Function.

```
Pkg.activate([s::String]; shared::Bool=false)
```

Activate the environment at `s`. The active environment is the environment that is modified by executing package commands. The logic for what path is activated is as follows:

- If `shared` is `true`, the first existing environment named `s` from the depots in the depot stack will be activated. If no such environment exists yet, activate it in the first depot.
- If `s` is a path that exist, that environment will be activated.
- If `s` is a package name in the current project activate that is tracking a path, activate the environment at that path.
- If `s` is a non-existing path, activate that path.

If no argument is given to `activate`, activate the home project, which is the one specified by either `--project` command line when starting julia, or JULIA_PROJECT environment variable.

Examples

```
Pkg.activate()
Pkg.activate("local/path")
Pkg.activate("MyDependency")
```

source

Pkg.rm – Function.

```
Pkg.rm(pkg::Union{String, Vector{String})
Pkg.rm(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Remove a package from the current project. If the `mode` of `pkg` is PKGMODE_MANIFEST also remove it from the manifest including all recursive dependencies of `pkg`.

See also PackageSpec, PackageMode.

source

Pkg.update – Function.

```
Pkg.update(; level::UpgradeLevel=UPLEVEL_MAJOR, mode::PackageMode = PKGMODE_PROJECT)
Pkg.update(pkg::Union{String, Vector{String})
Pkg.update(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Update a package pkg. If no posistional argument is given, update all packages in the manifest if mode is PKGMODE_MANIFEST and packages in both manifest and project if mode is PKGMODE_PROJECT. If no positional argument is given `level` can be used to control what how much packages are allowed to be upgraded (major, minor, patch, fixed).

See also PackageSpec, PackageMode, UpgradeLevel.

source

Pkg.test – Function.

```
Pkg.test(; coverage::Bool=true)
Pkg.test(pkg::Union{String, Vector{String}; coverage::Bool=true)
Pkg.test(pkgs::Union{PackageSpec, Vector{PackageSpec}}; coverage::Bool=true)
```

Run the tests for package pkg or if no positional argument is given to `test`, the current project is tested (which thus needs to be a package). A package is tested by running its `test/runtests.jl` file.

The tests are run by generating a temporary environment with only `pkg` and its (recursive) dependencies (recursively) in it. If a manifest exist, the versions in that manifest is used, otherwise a feasible set of package are resolved and installed.

During the test, test-specific dependencies are active, which are given in the project file as e.g.

```
[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = [Test]
```

Coverage statistics for the packages may be generated by passing `coverage=true`. The default behavior is not to run coverage.

The tests are executed in a new process with `check-bounds=yes` and by default `startup-file=no`. If using the startup file (`~/.julia/config/startup.jl`) is desired, start julia with `--startup-file=yes`.

source

Pkg.build – Function.

```
Pkg.build()
Pkg.build(pkg::Union{String, Vector{String})
Pkg.build(pkgs::Union{PackageSpec, Vector{PackageSpec}})
```

Run the build script in `deps/build.jl` for pkg and all of the dependencies in depth-first recursive order. If no argument is given to `build`, the current project is built, which thus needs to be a package. This function is called automatically one any package that gets installed for the first time.

source

Pkg.pin – Function.

```
Pkg.pin(pkg::Union{String, Vector{String})
Pkg.pin(pkgs::Union{Packagespec, Vector{Packagespec}})
```

Pin a package to the current version (or the one given in the `packagespec` or a certain git revision. A pinned package is never updated.

source

Pkg.free – Function.

```
Pkg.free(pkg::Union{String, Vector{String})
Pkg.free(pkgs::Union{Packagespec, Vector{Packagespec}})
```

Free a package which removes a `pin` if it exists, or if the package is tracking a path, e.g. after Pkg.develop, go back to tracking registered versions.

Examples

source

Pkg.instantiate – Function.

```
Pkg.instantiate()
```

If a `Manifest.toml` file exist in the current project, download all the packages declared in that manifest. Else, resolve a set of feasible packages from the `Project.toml` files and install them.

source

Pkg.resolve – Function.

```
Pkg.resolve()
```

Update the current manifest with eventual changes to the dependency graph from packages that are tracking a path.

source

Pkg.setprotocol! – Function.

```
Pkg.setprotocol!(proto::Union{Nothing, AbstractString}=nothing)
```

Set the protocol used to access GitHub-hosted packages when adding a url or `developing` a package. Defaults to 'https', with `proto == nothing` delegating the choice to the package developer.

source

Chapter 76

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.

Examples

source

Printf.@sprintf – Macro.

```
@sprintf("%Fmt", args...)
```

Return `@printf` formatted output as string.

Examples

source

Chapter 77

Profiling

Profile.@profile – Macro.

```
@profile
```

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

source

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.

source

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, whereas `:count` sorts in order of number of collected samples.
- `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 * `$\sqrt{N}$, 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.

source

```
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.

source

`Profile.init` – Function.

```
init(; n::Integer, delay::Float64)
```

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`).

source

`Profile.fetch` – Function.

```
fetch() -> data
```

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `clear`, can affect `data` unless you first make a copy. 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.

source

`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.

source

`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.

source

`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.

source

Chapter 78

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
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.6.0-dev.2493 (2017-01-31 18:53 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit c99e12c* (0 days old master)
|__/                   |  x86_64-linux-gnu

julia>
```

To exit the interactive session, type ^D – the control key together with the d key on a blank line – or type `quit()` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

78.1   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.

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.

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 `Base.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:

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.

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.

78.2   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`:

Users should refer to `LineEdit.jl` to discover the available actions on key input.

78.3 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:

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:

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:

Tab completion can help with investigation of the available methods matching the input arguments:

Keywords are also displayed in the suggested methods after ;, see below line where `limit` and `keepempty` are keyword arguments:

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:

Fields for output from functions can also be completed:

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

78.4 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:

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`:

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:

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)
meta-Backspace	Delete the previous word
meta-d	Forward delete the next word
^W	Delete previous text up to the nearest whitespace
meta-w	Copy the current region in the kill ring
meta-W	"Kill" the current region, placing the text in the kill ring
^K	"Kill" to end of line, placing the text in the kill ring
^Y	"Yank" insert the text from the kill ring
meta-y	Replace a previously yanked text with an older entry from the kill ring
^T	Transpose the characters about the cursor
meta-Up arrow	Transpose current line with line above
meta-Down arrow	Transpose current line with line below
meta-u	Change the next word to uppercase
meta-c	Change the next word to titlecase
meta-l	Change the next word to lowercase
^/, ^_	Undo previous editing action
^Q	Write a number in REPL and press ^Q to open editor at corresponding stackframe or method
meta-Left	indent the current line on the left

Chapter 79

TerminalMenus

TerminalMenus is a submodule of the Julia REPL and enables small, low-profile interactive menus in the terminal.

79.1 Examples

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.

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.

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
```

79.2   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

Chapter 80

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 81

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.

81.1 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:n` 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), and to $[0, 1)$ for floating point numbers;

S defaults to Float64 (except when dims is a tuple of integers, in which case S must be specified).

Examples

> **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.

source

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

source

Random.bitrand – Function.

```
bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a `BitArray` of random boolean values.

Examples

source

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.

Examples

source

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

source

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

source

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

source

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

> **Note**
>
> `chars` can be any collection of characters, of type `Char` or `UInt8` (more efficient), provided rand can randomly pick characters from it.

source

81.2 Subsequences, permutations and shuffling

Random.randsubseq – Function.

```
randsubseq(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.

source

Random.randsubseq! – Function.

```
randsubseq!(S, A, p)
```

Like randsubseq, but the results are stored in S (which is resized as needed).

source

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). To randomly permute an arbitrary vector, see shuffle or shuffle!.

Examples

source

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

source

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.

Examples

source

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

source

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

source

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

source

81.3   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.

Examples

source

`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

source

`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.

source

81.4 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

There are two categories: generating values from a type (e.g. `rand(Int)`), or from a collection (e.g. `rand(1:3)`). The simple cases are explained first, and more advanced usage is presented later. We assume here that the choice of algorithm is independent of the RNG, so we use `AbstractRNG` in our signatures.

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. In order to define random generation of values of type T, the following method can be defined: `rand(rng::AbstractRNG,` `::Random.SamplerType{T})` (this should return 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):

Scalar and array methods for `Die` now work as expected:

Generating values from a collection

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 order to define random generation out of objects of type S, the following method can 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:

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}})`

Optimizing generation with cached computation between calls

When repeatedly generating random values (with the same `rand` parameters), it happens for some types that the result of a computation is used for each call. In this case, the computation can be decoupled from actually generating the values. This is the case for example with the default implementation for `AbstractArray`. 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:

This mechanism is of course used by the default implementation of random array generation (like in `rand(1:20, 10)`). In order to implement this decoupling for a custom type, a helper type can be used. Going back to our `Die` example: `rand(::Die)` uses random generation from a range, so there is an opportunity for this optimization:

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.

This pattern is so frequent that a helper type named `Random.SamplerSimple` is available, saving us the definition of `SamplerDie`: we could have implemented our decoupling with:

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[]`.

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:

Of course, rand must also be defined on those types (i.e. `rand(::AbstractRNG, ::SamplerDie1)` and `rand(::AbstractRNG, ::SamplerDieMany)`).

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 82

SHA

Usage is very straightforward:

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:

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 83

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.

source

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.

source

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

source

Chapter 84

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.

source

Distributed.procs – Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

source

SharedArrays.sdata – *Function.*

```
sdata(S::SharedArray)
```

Returns the actual `Array` object backing S.

source

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.

source

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.

source

Chapter 85

Sockets

Sockets.connect – Method.

```
connect([host], port::Integer) -> TCPSocket
```

Connect to the host `host` on port `port`.

source

Sockets.connect – Method.

```
connect(path::AbstractString) -> PipeEndpoint
```

Connect to the named pipe / UNIX domain socket at `path`.

source

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.

source

Sockets.listen – Method.

```
listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

source

Sockets.getaddrinfo – Function.

```
getalladdrinfo(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.

source

Sockets.getipaddr – Function.

```
getipaddr() -> IPAddr
```

Get the IP address of the local machine.

Examples

source

`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

source

`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

source

`Sockets.getsockname` – Function.

```
getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr, UInt16)
```

Get the IP address and port that the given socket is bound to.

source

`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.

source

`Sockets.IPv4` – Type.

```
IPv4(host::Integer) -> IPv4
```

Returns an IPv4 object from ip address `host` formatted as an `Integer`.

source

`Sockets.IPv6` – Type.

```
IPv6(host::Integer) -> IPv6
```

Returns an IPv6 object from ip address `host` formatted as an `Integer`.

source

`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.

source

`Sockets.UDPSocket` – Type.

```
UDPSocket()
```

Open a UDP socket using libuv. `UDPSocket` has various fields to denote the state of the socket.

source

`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.

source

`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.

source

`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

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, 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) -> (address, data)
```

Read a UDP packet from the specified socket, returning a tuple of (`address`, `data`), where `address` will be either IPv4 or IPv6 as appropriate.

`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`).

Chapter 86

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.

86.1 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:

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.

86.2 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:

As for SparseMatrixCSC, the `SparseVector` type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

86.3   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:

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]`.

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).

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

You can go in the other direction using the `Array` constructor. The `issparse` function can be used to query if a matrix is sparse.

86.4   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.

86.5   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)`	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 87

Sparse Arrays

SparseArrays.SparseVector – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

source

SparseArrays.SparseMatrixCSC – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format.

source

SparseArrays.sparse – Function.

```
sparse(A)
```

Convert an AbstractMatrix A into a sparse matrix.

Examples

source

```
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 Base.SparseArrays.sparse!.

Examples

source

SparseArrays.sparsevec – Function.

```
sparsevec(I, V, [m, combine])
```

897

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

source

```
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

source

```
sparsevec(A)
```

Convert a vector A into a sparse vector of length m.

Examples

source

SparseArrays.issparse – Function.

```
issparse(S)
```

Returns `true` if S is sparse, and `false` otherwise.

Examples

source

SparseArrays.nnz – Function.

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

source

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

source

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

source

SparseArrays.spdiagm – Function.

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a square sparse diagonal matrix from `Pairs` of vectors and diagonals. Vector `kv.second` will be placed on the `kv.first` diagonal.

Examples

source

SparseArrays.blockdiag – Function.

```
blockdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

source

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

source

SparseArrays.sprandn – Function.

```
sprandn([rng], 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.

Examples

source

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

source

SparseArrays.rowvals – Function.

```
rowvals(A::SparseMatrixCSC)
```

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

source

SparseArrays.nzrange – Function.

```
nzrange(A::SparseMatrixCSC, 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 i = 1:n
   for j in nzrange(A, i)
      row = rows[j]
      val = vals[j]
      # perform sparse wizardry...
   end
end
```

source

SparseArrays.dropzeros! – Function.

```
dropzeros!(A::SparseMatrixCSC; trim::Bool = true)
```

Removes stored numerical zeros from A, optionally trimming resulting excess space from A.rowval and A.nzval when trim is true.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

source

```
dropzeros!(x::SparseVector; trim::Bool = true)
```

Removes stored numerical zeros from x, optionally trimming resulting excess space from x.nzind and x.nzval when trim is true.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

source

SparseArrays.dropzeros – Function.

```
dropzeros(A::SparseMatrixCSC; trim::Bool = true)
```

Generates a copy of A and removes stored numerical zeros from that copy, optionally trimming excess space from the result's rowval and nzval arrays when trim is true.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

```
dropzeros(x::SparseVector; trim::Bool = true)
```

Generates a copy of x and removes numerical zeros from that copy, optionally trimming excess space from the result's `nzind` and `nzval` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

`SparseArrays.permute` – Function.

```
permute(A::SparseMatrixCSC{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) == A.n`). Row-permutation p's length must match A's row count (`length(p) == A.m`).

For expert drivers and additional information, see `permute!`.

Examples

`Base.permute!` – Method.

```
permute!(X::SparseMatrixCSC{Tv,Ti}, A::SparseMatrixCSC{Tv,Ti},
        p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
        [C::SparseMatrixCSC{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::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
        q::AbstractVector{<:Integer}[, C::SparseMatrixCSC{Tv,Ti},
        [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

X's dimensions must match those of A (`X.m == A.m` and `X.n == A.n`), and X must have enough storage to accommodate all allocated entries in A (`length(X.rowval) >= nnz(A)` and `length(X.nzval) >= nnz(A)`). Column-permutation q's length must match A's column count (`length(q) == A.n`). Row-permutation p's length must match A's row count (`length(p) == A.m`).

C's dimensions must match those of `transpose(A)` (`C.m == A.n` and `C.n == A.m`), and C must have enough storage to accommodate all allocated entries in A (`length(C.rowval) >= nnz(A)` and `length(C.nzval) >= 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 88

Statistics

The Statistics module contains basic statistics functionality.

Statistics.std – Function.

```
std(v; corrected::Bool=true, mean=nothing, dims)
```

Compute the sample standard deviation of a vector or array `v`, optionally along the given dimensions. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sqrt(sum((v - mean(v)).^2) / (length(v) - 1))`. A pre-computed mean may be provided. 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 = `length(x)`.

> **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.

source

Statistics.stdm – Function.

```
stdm(v, m; corrected::Bool=true)
```

Compute the sample standard deviation of a vector `v` with known mean `m`. 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 = `length(x)`.

> **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.

source

Statistics.var – Function.

```
var(v; dims, corrected::Bool=true, mean=nothing)
```

Compute the sample variance of a vector or array v, optionally along the given dimensions. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of v is an IID drawn from that generative distribution. This computation is equivalent to calculating sum(abs2, v - mean(v)) / (length(v) - 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 = length(x). The mean mean over the region may be provided.

> **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.

source

Statistics.varm – Function.

```
varm(v, m; dims, corrected::Bool=true)
```

Compute the sample variance of a collection v with known mean(s) m, optionally over the given dimensions. m may contain means for each dimension of v. 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 = length(x).

> **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.

source

Statistics.cor – Function.

```
cor(x::AbstractVector)
```

Return the number one.

source

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix X along the dimension dims.

source

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors x and y.

source

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices X and Y along the dimension dims.

source

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)`.

source

```
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)`.

source

```
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 $\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})^*$.

source

```
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)`.

source

Statistics.mean! – Function.

```
mean!(r, v)
```

Compute the mean of v over the singleton dimensions of r, and write results to r.

Examples

source

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

source

```
mean(f::Function, itr)
```

Apply the function f to each element of collection `itr` and take the mean.

source

```
mean(A::AbstractArray; dims)
```

Compute the mean of an array over the given dimensions.

Examples

source

Statistics.median! – Function.

```
median!(v)
```

Like median, but may overwrite the input vector.

source

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

source

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

source

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.

source

```
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).

source

```
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.

source

```
middle(a)
```

Compute the middle of an array a, which consists of finding its extrema and then computing their mean.

source

Statistics.quantile! – Function.

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false)
```

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.

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 default.

> **Note**
>
> An `ArgumentError` is thrown if v contains NaN or missing values.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

Examples

source

Statistics.quantile – Function.

```
quantile(itr, p; sorted=false)
```

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.

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 default.

> **Note**
>
> An `ArgumentError` is thrown if `itr` contains NaN or missing values. Use the skipmissing function to omit `missing` entries and compute the quantiles of non-missing values.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

Examples

"'jldoctest 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 18.0

julia> quantile(skipmissing([1, 10, missing]), 0.5) 5.5 "'

source

Chapter 89

Unit Testing

89.1 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, [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 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

89.2 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

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:

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).

source

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

source

For example, suppose we want to check our new function `foo(x)` works as expected:

If the condition is true, a `Pass` is returned:

If the condition is false, then a `Fail` is returned and an exception is thrown:

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:

If we expect that evaluating an expression *should* throw an exception, then we can use `@test_throws` to check that this occurs:

89.3   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 `@testsets` regardless of their side-effect on the global RNG state.

Examples

source

We can put our tests for the `foo(x)` function in a test set:

Test sets can also be nested:

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:

89.4 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.

Test.@inferred – Macro.

```
@inferred 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.

source

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 `Symbols` may be used for the standard log levels, and `Regexs` 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)
```

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
```

source

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"
```

source

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.

source

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`.

source

89.5 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

source

`@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

source

89.6 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.

source

`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`.

source

`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.

> `source`

`Test.get_testset_depth` – Function.

> `get_testset_depth()`

Returns the number of active test sets, not including the default test set

> `source`

`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:

And using that testset looks like:

Chapter 90

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

source

`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

source

`UUIDs.uuid_version` – Function.

```
uuid_version(u::UUID) -> Int
```

Inspects the given UUID and returns its version (see RFC 4122).

Examples

source

Chapter 91

Unicode

– Function.

```
Unicode.isassigned(c) -> Bool
```

Returns `true` if the given char or integer is an assigned Unicode code point.

Examples

source

– Function.

```
Unicode.normalize(s::AbstractString, normalform::Symbol)
```

Normalize the string s according to one of the four "normal forms" of the Unicode standard: `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 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

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

Examples

source

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.)

source

Part VI

Developer Documentation

Chapter 92

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

92.1 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.

92.2 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:

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

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`:

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.

92.3 Subtypes

The *direct* subtypes of any `DataType` may be listed using `subtypes`. For example, the abstract `DataType` `AbstractFloat` has four (concrete) subtypes:

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.

92.4 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.

92.5   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.

92.6   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:

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:

92.7   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:

See @code_lowered, @code_typed, @code_warntype, @code_llvm, and @code_native.

Chapter 93

Documentation of Julia's Internals

93.1 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 over-
written.

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 environ-
ment 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()`.

93.2   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 lowered form, since it is more important to the compiler. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

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`	

Lowered form

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 method. 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 4-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 4-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.

- – `args[4]`

  `true` or `false`, identifying whether the method is staged (`@generated function`).

- `const`

  Declares a (global) variable as constant.

- `null`

  Has no arguments; simply yields the value `nothing`.

- `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.

- `return`

  Returns its argument as the value of the enclosing function.

- `the_exception`

  Yields the caught exception inside a `catch` block. This is the value of the run time system variable `jl_exception_in_transit`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `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`.

- `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 (usually compressed).

- `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.

- `min_world` / `max_world`

  The range of world ages for which this method is visible to dispatch.

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.

- `rettype`

  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 other information about the inference result such as a constant return value may be put here (if `jlcall_api == 2`), 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 '*jl*value*t* ()(*jl*function*t* *f, *jl*value*t* *args[nargs], uint32t nargs)'
    - 2 - Constant (value stored in `inferred`)
    - 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.

CodeInfo

A temporary container for holding lowered source code.

- `code`

  An Any array of statements

- `slotnames`

  An array of symbols giving the name of each slot (argument or local variable).

- `slottypes`

  An array of types for the slots.

- `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. If an array, specifies a type for each location.

- `linetable`

  An array of source location objects

- `codelocs`

  An array of integer indices into the `linetable`, giving the location associated with each statement.

Boolean properties:

- `inferred`

  Whether this has been produced by type inference.

- `inlineable`

  Whether this should be inlined.

- `propagate_inbounds`

  Whether this should 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.

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

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))`

do syntax:

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 juliaparser.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:

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block 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)

Imports and such

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:

parses as:

```
(if a (block (line 2) b)
```

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....")`

```
    (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:

parses as:

```
(function (where (call f (parameters (kw k 1))
                       (:: x T))
                 T)
          (block (line 2) (return (call + x 1))))
```

Type definition:

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 `LineNumberNodes` 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.

93.3   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`:

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, :obj:`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:

The signature of `f3` is a `UnionAll` type wrapping a tuple type. 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:

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, `TypeVars` 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 `TypeVars` manually:

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

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:

These can be distinguished by examining the name field of the type, which is an object of type `TypeName`:

In this case, the relevant field is `wrapper`, which holds a reference to the top-level type used to make new `Array` types.

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:

(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:

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

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:

What is the "primary" tuple-type?

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:

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:

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

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.

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:

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:

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:

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:

We can make it more interesting by trying a more complex case:

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:

which allows you to test whether tuple type a is more specific than tuple type b.

93.4 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);
jl_uniontype_t *jl_new_uniontype(jl_tuple_t *types);
```

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

93.5 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.

93.6   Calling Conventions

Julia uses three calling conventions for four distinct purposes:

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

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.

93.7 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.

93.8   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 TypeNames. 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:

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:

is lowered to (roughly):

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:

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:

The second method is an ordinary definition for the original `circle` function, which handles the case where no keyword arguments are passed:

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:

is lowered to:

`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.

93.9 Base.Cartesian

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. It is hoped that Cartesian will not, in the long term, be necessary; however, at present it is one of the few ways to write compact and performant multidimensional code.

Principles of usage

A simple example of usage is:

which generates the following code:

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 `1:size(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 = 1:size(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`:

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. If you're writing code that you need to work with older Julia versions, currently you should use the `@ngenerate` macro described in an older version of this documentation.

Starting in Julia 0.4-pre, the recommended approach is to use a `@generated function`. Here's an example:

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` generates n expressions that follow a pattern. This code would generate the following statements:

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:

would generate expressions

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

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

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

93.10   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:

Here, `ex` is expected to be an expression defining a function. A statement like this:

gets turned into an expression like this:

`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++.

93.11   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.

Indexing: cartesian vs. linear indexing

Broadly speaking, there are two main ways to access data in an array. The first, often called cartesian indexing, uses N indices for an N -dimensional `AbstractArray`. For example, a matrix A (2-dimensional) can be indexed in cartesian style as `A[i,j]`. The second indexing method, referred to as linear indexing, uses a single index even for higher-dimensional objects. For example, if `A = reshape(1:12, 3, 4)`, then the expression `A[5]` returns the value 5. Julia allows you to combine these styles of indexing: for example, a 3d array A3 can be indexed as `A3[i,j]`, in which case `i` is interpreted as a cartesian index for the first dimension, and `j` is a linear index over dimensions 2 and 3.

For `Arrays`, linear indexing appeals to the underlying storage format: an array is laid out as a contiguous block of memory, and hence the linear index is just the offset (+1) of the corresponding entry relative to the beginning of the array. However, this is not true for many other `AbstractArray` types: examples include [SparseMatrixCSC](#) from the `SparseArrays` standard library module, arrays that require some kind of computation (such as interpolation), and the type under discussion here, `SubArray`. For these types, the underlying information is more naturally described in terms of cartesian indices.

The `getindex` and `setindex!` functions for `AbstractArray` types may include automatic conversion between indexing types. For explicit conversion, [CartesianIndices](#) can be used.

While converting from a cartesian index to a linear index is fast (it's just multiplication and addition), converting from a linear index to a cartesian index is very slow: it relies on the `div` operation, which is one of the slowest low-level operations you can perform with a CPU. For this reason, any code that deals with `AbstractArray` types is best designed in terms of cartesian, rather than linear, indexing.

Index replacement

Consider making 2d slices of a 3d array:

`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:

`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.Sli` Note in particular the tuple parameter, which stores the types of the indices used to create S1. Likewise,

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:

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

for any pair of indices (`i,j`) (except `CartesianIndex`s 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:

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:

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,

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:

  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:

  Instead, though, we get:

  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.

93.12   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 16 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 16 bytes of the field in the structure's memory; a `0x01` value would indicate that a `UInt8` value was stored in the first 8 bytes of the 16 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: it's 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 it's 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.

93.13   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.

Julia now ships with a script that automates the tasks of building the system image, wittingly named `build_sysimg.jl` that lives in `DATAROOTDIR/julia/`. That is, to include it into a current Julia session, type:

This will include a `build_sysimg` function:

Main.BuildSysImg.build_sysimg – Function.

```
build_sysimg(sysimg_path=default_sysimg_path(), cpu_target="native", userimg_path=nothing;
    force=false)
```

Rebuild the system image. Store it in `sysimg_path`, which defaults to a file named `sys.ji` that sits in the same folder as `libjulia.{so,dylib}`, except on Windows where it defaults to `Sys.BINDIR/../lib/julia/sys.ji`. Use the cpu instruction set given by `cpu_target`. Valid CPU targets are the same as for the `-C` option to `julia`, or the `-march` option to `gcc`. Defaults to `native`, which means to use all CPU instructions available on the current processor. Include the user image file given by `userimg_path`, which should contain directives such as `using MyPackage` to include that package in the new system image. New system image will not replace an older image unless `force` is set to true.

source

Note that this file can also be run as a script itself, with command line arguments taking the place of arguments passed to the `build_sysimg` function. For example, to build a system image in `/tmp/sys.{so,dll,dylib}`, with the `core2` CPU instruction set, a user image of `~/userimg.jl` and `force` set to `true`, one would execute:

```
julia build_sysimg.jl /tmp/sys core2 ~/userimg.jl --force
```

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 features.

Specifying multiple system image targets

Multi-microarch 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. The syntax for each target is a CPU name followed by multiple features separated by `,`. All features supported by LLVM is 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 nth target (0-based) will be used as the base target instead of the default (0th) one. The base target has to be either 0 or another `clone_all` target. Specifying a non default `clone_all` target as the base target will cause an error.

3. `opt_size`

   This cause the function for the targe to be optimize for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

4. `min_size`

   This cause the function for the targe to be optimize for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

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`.

93.14   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/`.

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

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 = 3.5.0
```

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:

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

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:

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:

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 cor-
responding 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.

93.15   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 ob-
jects 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);
}
```

93.16 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:

With a custom array-like type MyArray having:

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 @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,

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).

93.17   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
- gc*perm*lock
- 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 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 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

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance updates : codegen lock

- These fields are generally lazy initialized, using the test-and-test-and-set pattern.
- These are set at construction and immutable:
    - specTypes
    - sparam_vals
    - def
- These are set by `jl_type_infer` (while holding codegen lock):
    - rettype
    - inferred

- – these can also be reset, see `jl_set_lambda_rettype` for that logic as it needs to keep `functionObjectsDecls` in sync

- `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 (`jlcall_api` and `fptr`, `unspecialized_ducttape`):

  - – 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

- `compile_traced` flag:

  - – unknown

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)

- invoke / specializations / tfunc modifications

93.18   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

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}(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:

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 multidimensional. 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!`:

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}(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`:

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)`:

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:

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`:

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

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.

93.19   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.

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:

93.20   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))`:

If your debugging adventures require a `MethodInstance`, you can look it up by calling `Core.Compiler.code_for_method` using many of the variables above. A `CodeInfo` object may be obtained with

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 this example:

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.

Chapter 94

Developing/debugging Julia's C code

94.1   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()` in any report you create:

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:

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.....

94.2 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

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

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.

94.3   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`:

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.

94.4 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`.