
Julia Language Documentation

Release 0.3.6-pre

Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al.

February 05, 2015

1	The Julia Manual	1
1.1	Introduction	1
1.2	Getting Started	2
1.3	Variables	4
1.4	Integers and Floating-Point Numbers	6
1.5	Mathematical Operations and Elementary Functions	16
1.6	Complex and Rational Numbers	24
1.7	Strings	29
1.8	Functions	40
1.9	Control Flow	48
1.10	Scope of Variables	62
1.11	Types	67
1.12	Methods	83
1.13	Constructors	91
1.14	Conversion and Promotion	98
1.15	Modules	103
1.16	Metaprogramming	106
1.17	Multi-dimensional Arrays	118
1.18	Linear algebra	128
1.19	Networking and Streams	130
1.20	Parallel Computing	134
1.21	Interacting With Julia	143
1.22	Running External Programs	147
1.23	Calling C and Fortran Code	152
1.24	Embedding Julia	159
1.25	Packages	163
1.26	Package Development	169
1.27	Profiling	175
1.28	Memory allocation analysis	179
1.29	Performance Tips	180
1.30	Style Guide	189
1.31	Frequently Asked Questions	193
1.32	Noteworthy Differences from other Languages	206
1.33	Unicode Input	209
2	The Julia Standard Library	211
2.1	Essentials	211
2.2	Collections and Data Structures	223

2.3	Mathematics	234
2.4	Numbers	252
2.5	Strings	259
2.6	Arrays	264
2.7	Tasks and Parallel Computing	272
2.8	Linear Algebra	277
2.9	Constants	289
2.10	Filesystem	290
2.11	I/O and Network	293
2.12	Punctuation	303
2.13	Sorting and Related Functions	304
2.14	Package Manager Functions	307
2.15	Graphics	309
2.16	Unit and Functional Testing	311
2.17	Testing Base Julia	313
2.18	C Interface	313
2.19	Profiling	317
	Bibliography	319

The Julia Manual

1.1 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; 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

1.2 Getting Started

Julia installation is straightforward, whether using precompiled binaries or compiling from source. Download and install Julia by following the instructions at <http://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”):

```
$ julia

      _       _ _(_)_      | A fresh approach to technical computing
  (_)_       | (_)_(_)    | Documentation: http://docs.julialang.org
    _ _     _||_ _ _ _    | Type "help()" to list help topics
  || || || || ||/_ _'|   |
  || ||_|| || |(_|| |   | Version 0.3.0-prerelease+3690 (2014-06-16 05:11 UTC)
_/_|_\'_||_||_||_\'_||   | Commit 1b73f04* (0 days old master)
|_/_/                  | x86_64-apple-darwin13.1.0

julia> 1 + 2
3

julia> ans
3
```

To exit the interactive session, type `^D` — the control key together with the `d` key or type `quit()`. 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 taken as command-line arguments to the program `script.jl`, passed in the global constant `ARGS`. `ARGS` is also set when script code is given using the `-e` option on the command line (see the `julia help` output below). For example, to just print the arguments given to a script, you could do this:

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

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

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

Julia can be started in parallel mode with either the `-p` or the `--machinefile` options. `-p n` will launch an additional `n` worker processes, while `--machinefile file` will launch a worker for each line in file `file`. The machines defined in `file` must be accessible via a passwordless `ssh` login, with Julia installed at the same location as the current host. Each machine definition takes the form `[user@]host[:port] [bind_addr].user` defaults to current user, `port` to the standard `ssh` port. Optionally, in case of multi-homed hosts, `bind_addr` may be used to explicitly specify an interface.

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

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

...
```

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

```
julia [options] [program] [args...]
-v, --version          Display version information
-h, --help             Print this message
-q, --quiet            Quiet startup without banner
-H, --home <dir>      Set location of julia executable

-e, --eval <expr>      Evaluate <expr>
-E, --print <expr>     Evaluate and show <expr>
-P, --post-boot <expr> Evaluate <expr> right after boot
-L, --load <file>       Load <file> right after boot on all processors
-J, --sysimage <file>  Start up with the given system image file

-p <n>                 Run n local processes
```

```
--machinefile <file>      Run processes on hosts listed in <file>

-i                          Force isinteractive() to be true
--no-history-file          Don't load or save history
-f, --no-startup           Don't load ~/.juliarc.jl
-F                          Load ~/.juliarc.jl, then handle remaining inputs
--color={yes|no}           Enable or disable color text

--code-coverage            Count executions of source lines
--check-bounds={yes|no}    Emit bounds checks always or never (ignoring declarations)
--int-literals={32|64}     Select integer literal size independent of platform
```

1.2.1 Resources

In addition to this manual, there are various other resources that may help new users get started with Julia:

- [Julia and IJulia cheatsheet](#)
- [Learn Julia in a few minutes](#)
- [Tutorial for Homer Reid's numerical analysis class](#)
- [An introductory presentation](#)
- [Videos from the Julia tutorial at MIT](#)
- [Forio Julia Tutorials](#)

1.3 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 = "人人生而自由，在尊严和权力上一律平等。"
"人人生而自由，在尊严和权力上一律平等。"
```

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

```
julia> δ = 0.00001
1.0e-5

julia> 안녕하세요 = "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`. Julia will even let you redefine built-in constants and functions if needed:

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
Warning: imported binding for pi overwritten in module Main
3

julia> pi
3

julia> sqrt(100)
10.0

julia> sqrt = 4
Warning: imported binding for sqrt overwritten in module Main
4
```

However, this is obviously not recommended to avoid potential confusion.

1.3.1 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 \oplus , are parsed as infix operators and are available for user-defined methods (e.g. you can use `const ⊗ = kron` to define \otimes as an infix Kronecker product).

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 "="
```

1.3.2 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` 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 modify their inputs have names that end in `!`. These functions are sometimes called mutating functions or in-place functions.

1.4 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
<code>Int8</code>	x	8	-2^7	$2^7 - 1$
<code>UInt8</code>		8	0	$2^8 - 1$
<code>Int16</code>	x	16	-2^{15}	$2^{15} - 1$
<code>UInt16</code>		16	0	$2^{16} - 1$
<code>Int32</code>	x	32	-2^{31}	$2^{31} - 1$
<code>UInt32</code>		32	0	$2^{32} - 1$
<code>Int64</code>	x	64	-2^{63}	$2^{63} - 1$
<code>UInt64</code>		64	0	$2^{64} - 1$
<code>Int128</code>	x	128	-2^{127}	$2^{127} - 1$
<code>UInt128</code>		128	0	$2^{128} - 1$
<code>Bool</code>	N/A	8	false (0)	true (1)
<code>Char</code>	N/A	32	'\0'	'\Uffffffff'

`Char` natively supports representation of [Unicode characters](#); see [Strings](#) for more details.

- **Floating-point types:**

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

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*.

1.4.1 Integers

Literal integers are represented in the standard manner:

```
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:

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

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

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

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

# 64-bit system:
julia> 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.:

```
# 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
```

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> 0o10
0x08

julia> typeof(ans)
UInt8
```

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 = {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.

To minimize the practical impact of this overflow, integer addition, subtraction, multiplication, and exponentiation operands are promoted to `Int` or `UInt` from narrower integer types. (However, divisions, remainders, and bitwise operations do not promote narrower types.)

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.

1.4.2 Floating-Point Numbers

Literal floating-point numbers are represented in the standard formats:

```
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:

```
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 only as a storage format. In calculations they'll be converted to `Float32`:

```
julia> sizeof(float16(4.))
2

julia> 2*float16(4.)
8.0f0
```

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 `bits` function:

```
julia> 0.0 == -0.0
true

julia> bits(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

julia> bits(-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:

Special value			Name	Description
Float16	Float32	Float64		
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> 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> (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> prevfloat(x)
1.2499999f0

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

julia> bits(x)
"00111111101000000000000000000000"

julia> bits(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, if wanted, the manner in which this rounding is done can be changed according to the rounding modes presented in the [IEEE 754 standard](#):

```
julia> 1.1 + 0.1
1.2000000000000002

julia> with_rounding(Float64, RoundDown) do
    1.1 + 0.1
end
1.2
```

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](#).

1.4.3 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, or from `String`. Once created, they participate in arithmetic with all other numeric types thanks to Julia's *type promotion and conversion mechanism*. :

[illegible]

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

```
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 (constructor with 10 methods)
```

The default precision (in number of bits of the significand) and rounding mode of `BigFloat` operations can be changed, and all further calculations will take these changes in account:

[illegible]

1.4.4 Numeric Literal Coefficients

To make common numeric formulas 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 the same as that of unary operators such as negation. So $2^3 \times$ is parsed as $2^3 \times$, and 2×3 is parsed as $2 \times (3)$.

Numeric literals also work as coefficients to parenthesized expressions:

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

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: type: apply: expected Function, got Int64
```

```
julia> x(x+1)
ERROR: type: apply: expected Function, got Int64
```

Both of these 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.

In both cases, we resolve the ambiguity in favor of interpretation as a 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.

1.4.5 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
<code>zero(x)</code>	Literal zero of type <code>x</code> or type of variable <code>x</code>
<code>one(x)</code>	Literal one of type <code>x</code> or type of variable <code>x</code>

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)
1e+00 with 256 bits of precision
```

1.5 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.

1.5.1 Arithmetic Operators

The following [arithmetic operators](#) are supported on all primitive numeric types:

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

as well as the negation on `Bool` types:

Expression	Name	Description
<code>!x</code>	negation	changes <code>true</code> to <code>false</code> 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 less tightly binding operators less tightly, but there are no syntactic constraints.)

1.5.2 Bitwise Operators

The following [bitwise operators](#) are supported on all primitive integer types:

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

Here are some examples with bitwise operators:

```
julia> ~123
-124
```

```
julia> 123 & 234
106
```

```
julia> 123 | 234
251
```

```
julia> 123 $ 234
145

julia> ~uint32(123)
0xffffffff84

julia> ~uint8(123)
0x84
```

1.5.3 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> 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> x = 0x01; typeof(x)
UInt8

julia> x *= 2 #Same as x = x * 2
2

julia> isa(x, Int)
true
```

1.5.4 Numeric Comparisons

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

Operator	Name
<code>==</code>	equality
<code>!=</code> \neq	inequality
<code><</code>	less than
<code><=</code> \leq	less than or equal to
<code>></code>	greater than
<code>>=</code> \geq	greater than or equal to

Here are some simple examples:

```
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 than 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> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

and can cause especial headaches with *Arrays*:

```
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
<code>isequal(x, y)</code>	<code>x</code> and <code>y</code> are identical
<code>isfinite(x)</code>	<code>x</code> is a finite number
<code>isinf(x)</code>	<code>x</code> is infinite
<code>isnan(x)</code>	<code>x</code> is not a number

`isequal()` considers NaNs equal to each other:

```
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> -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> 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.

The operator `<` is intended for array objects; the operation `A .< B` is valid only if `A` and `B` have the same dimensions. The operator returns an array with boolean entries and with the same dimensions as `A` and `B`. Such operators are called *elementwise*; Julia offers a suite of elementwise operators: `*`, `+`, etc. Some of the elementwise operators can take a scalar operand such as the example `0 .< A .< 1` in the preceding paragraph. This notation means that the scalar operand should be replicated for each entry of the array.

Note the evaluation behavior of chained comparisons:


```

v(x) = (println(x); x)

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](#)).

Operator Precedence

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

Category	Operators
Syntax	<code>.</code> followed by <code>:</code>
Exponentiation	<code>^</code> and its elementwise equivalent <code>.^</code>
Fractions	<code>//</code> and <code>./</code>
Multiplication	<code>*</code> <code>/</code> <code>%</code> <code>&</code> <code>\</code> and <code>.*</code> <code>./</code> <code>.*</code> <code>.\</code>
Bitshifts	<code><<</code> <code>>></code> <code>>>></code> and <code>.<<</code> <code>.>></code> <code>.>>></code>
Addition	<code>+</code> <code>-</code> <code> </code> <code>\$</code> and <code>.+</code> <code>.-</code>
Syntax	<code>:</code> <code>..</code> followed by <code> ></code>
Comparisons	<code>></code> <code><</code> <code>>=</code> <code><=</code> <code>==</code> <code>===</code> <code>!=</code> <code>!==</code> <code><:</code> and <code>.></code> <code>.<</code> <code>.>=</code> <code>.<=</code> <code>==</code> <code>!=</code>
Control flow	<code>&&</code> followed by <code> </code> followed by <code>?</code>
Assignments	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>//=</code> <code>\=</code> <code>^=</code> <code>%=</code> <code> =</code> <code>&=</code> <code>\$=</code> <code><<=</code> <code>>>=</code> <code>>>>=</code> and <code>.*=</code> <code>./=</code> <code>./.=</code> <code>.\=</code> <code>.^=</code> <code>.*=</code>

1.5.5 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 complexes, wherever such definitions make sense.

Rounding functions

Function	Description	Return type
<code>round(x)</code>	round x to the nearest integer	<code>FloatingPoint</code>
<code>iround(x)</code>	round x to the nearest integer	<code>Integer</code>
<code>floor(x)</code>	round x towards $-\text{Inf}$	<code>FloatingPoint</code>
<code>ifloor(x)</code>	round x towards $-\text{Inf}$	<code>Integer</code>
<code>ceil(x)</code>	round x towards $+\text{Inf}$	<code>FloatingPoint</code>
<code>iceil(x)</code>	round x towards $+\text{Inf}$	<code>Integer</code>
<code>trunc(x)</code>	round x towards zero	<code>FloatingPoint</code>
<code>itrunc(x)</code>	round x towards zero	<code>Integer</code>

Division functions

Function	Description
<code>div(x, y)</code>	truncated division; quotient rounded towards zero
<code>fld(x, y)</code>	floored division; quotient rounded towards $-\text{Inf}$
<code>rem(x, y)</code>	remainder; satisfies $x == \text{div}(x, y) * y + \text{rem}(x, y)$; sign matches x
<code>divrem(x, y)</code>	returns $(\text{div}(x, y), \text{rem}(x, y))$
<code>mod(x, y)</code>	modulus; satisfies $x == \text{fld}(x, y) * y + \text{mod}(x, y)$; sign matches y
<code>mod2pi(x)</code>	modulus with respect to 2π ; $0 \leq \text{mod2pi}(x) < 2\pi$
<code>gcd(x, y...)</code>	greatest common divisor of x, y, \dots ; sign matches x
<code>lcm(x, y...)</code>	least common multiple of x, y, \dots ; sign matches x

Sign and absolute value functions

Function	Description
<code>abs(x)</code>	a positive value with the magnitude of x
<code>abs2(x)</code>	the squared magnitude of x
<code>sign(x)</code>	indicates the sign of x , returning -1, 0, or +1
<code>signbit(x)</code>	indicates whether the sign bit is on (true) or off (false)
<code>copysign(x, y)</code>	a value with the magnitude of x and the sign of y
<code>flipsign(x, y)</code>	a value with the magnitude of x and the sign of $x*y$

Powers, logs and roots

Function	Description
<code>sqrt(x)</code> \sqrt{x}	square root of x
<code>cbrrt(x)</code> $\sqrt[3]{x}$	cube root of x
<code>hypot(x, y)</code>	hypotenuse of right-angled triangle with other sides of length x and y
<code>exp(x)</code>	natural exponential function at x
<code>expm1(x)</code>	accurate $\exp(x) - 1$ for x near zero
<code>ldexp(x, n)</code>	$x * 2^n$ computed efficiently for integer values of n
<code>log(x)</code>	natural logarithm of x
<code>log(b, x)</code>	base b logarithm of x
<code>log2(x)</code>	base 2 logarithm of x
<code>log10(x)</code>	base 10 logarithm of x
<code>log1p(x)</code>	accurate $\log(1+x)$ for x near zero
<code>exponent(x)</code>	binary exponent of x
<code>significand(x)</code>	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     atan2
```

These are all single-argument functions, with the exception of `atan2`, which gives the angle in radians between the x -axis and the point specified by its arguments, interpreted as x and y coordinates.

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

Function	Description
<code>erf(x)</code>	error function at x
<code>erfc(x)</code>	complementary error function, i.e. the accurate version of $1 - \text{erf}(x)$ for large x
<code>erfinv(x)</code>	inverse function to <code>erf()</code>
<code>erfcinv(x)</code>	inverse function to <code>erfc()</code>
<code>erfi(x)</code>	imaginary error function defined as $-im * \text{erf}(x * im)$, where im is the imaginary unit
<code>erfcx(x)</code>	scaled complementary error function, i.e. accurate $\exp(x^2) * \text{erfc}(x)$ for large x
<code>dawson(x)</code>	scaled imaginary error function, a.k.a. Dawson function, i.e. accurate $\exp(-x^2) * \text{erfi}(x)$ for large x
<code>gamma(x)</code>	gamma function at x
<code>lgamma(x)</code>	accurate $\log(\text{gamma}(x))$ for large x
<code>lfact(x)</code>	accurate $\log(\text{factorial}(x))$ for large x ; same as <code>lgamma(x+1)</code> for large x
<code>digamma(x)</code>	digamma function (i.e. the derivative of <code>lgamma()</code>) at x
<code>beta(x, y)</code>	beta function at x, y
<code>lbeta(x, y)</code>	accurate $\log(\text{beta}(x, y))$ for large x or y
<code>eta(x)</code>	Dirichlet eta function at x
<code>zeta(x)</code>	Riemann zeta function at x
<code>airy(z), airyai(z), airy(0, z)</code>	Airy Ai function at z
<code>airyprime(z), airyaiprime(z), airy(1, z)</code>	derivative of the Airy Ai function at z
<code>airybi(z), airy(2, z)</code>	Airy Bi function at z
<code>airybiprime(z), airy(3, z)</code>	derivative of the Airy Bi function at z
<code>airyx(z), airyx(k, z)</code>	scaled Airy AI function and k th derivatives at z
<code>besselj(nu, z)</code>	Bessel function of the first kind of order nu at z
<code>besselj0(z)</code>	<code>besselj(0, z)</code>
<code>besselj1(z)</code>	<code>besselj(1, z)</code>
<code>besseljx(nu, z)</code>	scaled Bessel function of the first kind of order nu at z

Table 1.1 – continued from previous page

Function	Description
<code>bessely(nu, z)</code>	Bessel function of the second kind of order <code>nu</code> at <code>z</code>
<code>bessely0(z)</code>	<code>bessely(0, z)</code>
<code>bessely1(z)</code>	<code>bessely(1, z)</code>
<code>besselyx(nu, z)</code>	scaled Bessel function of the second kind of order <code>nu</code> at <code>z</code>
<code>besselh(nu, k, z)</code>	Bessel function of the third kind (a.k.a. Hankel function) of order <code>nu</code> at <code>z</code> ; <code>k</code>
<code>hankelh1(nu, z)</code>	<code>besselh(nu, 1, z)</code>
<code>hankelh1x(nu, z)</code>	scaled <code>besselh(nu, 1, z)</code>
<code>hankelh2(nu, z)</code>	<code>besselh(nu, 2, z)</code>
<code>hankelh2x(nu, z)</code>	scaled <code>besselh(nu, 2, z)</code>
<code>besseli(nu, z)</code>	modified Bessel function of the first kind of order <code>nu</code> at <code>z</code>
<code>besselix(nu, z)</code>	scaled modified Bessel function of the first kind of order <code>nu</code> at <code>z</code>
<code>besselk(nu, z)</code>	modified Bessel function of the second kind of order <code>nu</code> at <code>z</code>
<code>besselkx(nu, z)</code>	scaled modified Bessel function of the second kind of order <code>nu</code> at <code>z</code>

1.6 Complex and Rational Numbers

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

1.6.1 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> 1 + 2im
1 + 2im
```

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

```
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
2.7296244647840084 - 6.960664459571898im
```

```
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> real(1 + 2im)
1

julia> imag(1 + 2im)
2

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

julia> abs(1 + 2im)
2.23606797749979

julia> abs2(1 + 2im)
5

julia> angle(1 + 2im)
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
sqrt will only return a complex result if called with a complex argument.
try sqrt(complex(x))
in sqrt at math.jl:131

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

The *literal numeric coefficient notation* does not work when constructing 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> 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
```

1.6.2 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 `num()` and `den()` functions:

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

```
julia> den(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> 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} (constructor with 1 method)
```

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

```
julia> 0//0
ERROR: invalid rational: 0//0
   in Rational at rational.jl:6
   in // at rational.jl:15
```

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
```


1.7 Strings

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

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

- `String` is an abstraction, not a concrete type — many different representations can implement the `String` interface, but they can easily be used together and interact transparently. Any string type can be used in any function expecting a `String`.
- Like C and Java, but unlike most dynamic languages, Julia has a first-class type representing a single character, called `Char`. This is just a special kind of 32-bit integer whose numeric value represents a Unicode code point.
- As in Java, strings are immutable: the value of a `String` 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.
- Julia supports the full range of [Unicode](#) characters: literal strings are always [ASCII](#) or [UTF-8](#) but other encodings for strings from external sources can be supported.

1.7.1 Characters

A `Char` value represents a single character: it is just a 32-bit integer with a special literal representation and appropriate arithmetic behaviors, whose numeric value is interpreted as a [Unicode code point](#). Here is how `Char` values are input and shown:

```
julia> 'x'
'x'

julia> typeof(ans)
Char
```

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

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

julia> typeof(ans)
Int64
```

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

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

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 `is_valid_char()` function:

```
julia> char(0x110000)
'\U110000'
```

```
julia> is_valid_char(0x110000)
false
```

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

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

```
julia> '\u0'
'\0'
```

```
julia> '\u78'
'x'
```

```
julia> '\u2200'
'∀'
```

```
julia> '\U10ffff'
'\U10ffff'
```

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

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

```
julia> int('\t')
9
```

```
julia> int('\n')
10
```

```
julia> int('\e')
27
```

```
julia> int('\x7f')
127
```

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

```
julia> int('\xff')
255
```

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

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

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

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

julia> 'x' - 'a'
23

julia> 'A' + 1
'B'
```

1.7.2 String Basics

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

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

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

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

```
julia> str[1]
'H'

julia> str[6]
','

julia> str[end]
'\n'
```

All indexing in Julia is 1-based: the first element of any integer-indexed object is found at index 1, and the last element is found at index `n`, when the string has a length of `n`.

In any indexing expression, the keyword `end` can be used as a shorthand for the last index (computed by `endof(str)`). You can perform arithmetic and other operations with `end`, just like a normal value:

```
julia> str[end-1]
'.'

julia> str[end/2]
','

julia> str[end/3]
ERROR: InexactError()
  in getindex at string.jl:59

julia> str[end/4]
ERROR: InexactError()
  in getindex at string.jl:59
```

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

```
julia> str[0]
ERROR: BoundsError()
  in getindex at <julia root>/usr/lib/julia/sys.dylib (repeats 2 times)

julia> str[end+1]
ERROR: BoundsError()
  in getindex at <julia root>/usr/lib/julia/sys.dylib (repeats 2 times)
```

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]
','

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.

1.7.3 Unicode and UTF-8

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

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

Whether these Unicode characters are displayed as escapes or shown as special characters depends on your terminal's locale settings and its support for Unicode. Non-ASCII 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> s[1]
'∀'

julia> s[2]
ERROR: invalid UTF-8 character index
  in next at ./utf8.jl:68
  in getindex at string.jl:57

julia> s[3]
ERROR: invalid UTF-8 character index
  in next at ./utf8.jl:68
  in getindex at string.jl:57

julia> s[4]
', '
```

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.

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 `endof(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) <= endof(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 = 1:endof(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

UTF-8 is not the only encoding that Julia supports, and adding support for new encodings is quite easy. In particular, Julia also provides `UTF16String` and `UTF32String` types, constructed by `utf16()` and `utf32()` respectively, for UTF-16 and UTF-32 encodings. It also provides aliases `WString` and `wstring()` for either UTF-16 or UTF-32 strings, depending on the size of `Cwchar_t`. 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, which goes into some greater detail.

1.7.4 Interpolation

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

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

```
julia> whom = "world"
"world"
```

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

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

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

This is more readable and convenient and equivalent to the above string concatenation — the system rewrites this apparent single string literal into 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> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

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

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

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

`string()` is the identity for String and Char values, so these are interpolated into strings as themselves, unquoted and unescaped:

```
julia> c = 'x'
'x'

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

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

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

1.7.5 Common Operations

You can lexicographically compare strings using the standard comparison operators:

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

julia> "abracadabra" == "xylophone"
false

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

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

You can search for the index of a particular character using the `search()` function:

1.7.6 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.

1.7.7 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> r"^s*(?:#|$)"
r"^s*(?:#|$)"

julia> typeof(ans)
Regex {constructor with 3 methods}
```

To check if a regex matches a string, use `ismatch()`:

```
julia> ismatch(r"^s*(?:#|$)", "not a comment")
false

julia> ismatch(r"^s*(?:#|$)", "# a comment")
true
```

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

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

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

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

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

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


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

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

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

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

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

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

- the entire substring matched: `m.match`
- the captured substrings as a tuple 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's is a pair of somewhat contrived examples:

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

```
julia> m.match
"acd"
```

```
julia> m.captures
3-element Array{Union{SubString{UTF8String}, Nothing}, 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{SubString{UTF8String}, Nothing}, 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 a tuple so that one can use tuple destructuring syntax to bind them to local variables:

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

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")
```

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

1.7.8 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 literal byte arrays — i.e. arrays of `UInt8` values. The convention is that non-standard literals with uppercase prefixes produce actual string objects, while those with lowercase prefixes produce non-string objects like byte arrays or compiled regular expressions. 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 Array{UInt8,1}:
 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 — if you try to use this as a regular string literal, you will get a syntax error:

```
julia> "DATA\xff\u2200"
ERROR: syntax: invalid UTF-8 sequence
```

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

```
julia> b"\xff"
1-element Array{UInt8,1}:
 0xff

julia> b"\uff"
2-element Array{UInt8,1}:
 0xc3
 0xbf
```

In character literals, this distinction is glossed over and `\xff` is allowed to represent the code point 255, because characters *always* represent code points. In strings, however, `\x` escapes always represent bytes, not code points, whereas `\u` and `\U` escapes always represent code points, which are encoded in one or more bytes. 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.

1.7.9 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 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.

1.8 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:

```
function f(x, y)
    x + y
end
```

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”:

```
f(x, y) = x + y
```

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> f(2,3)
5
```

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

```
julia> g = f;

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

There are two other ways that functions can be applied: using special operator syntax for certain function names (see Operators Are Functions below), or with the `apply()` function:

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

`apply()` applies its first argument — a function object — to its remaining arguments.

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

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

1.8.1 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.

1.8.2 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:

```
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:

```
f(x,y) = x + y

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

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:

```
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
```

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.

1.8.3 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.

1.8.4 Operators With Special Names

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

Expression	Calls
[A B C ...]	<code>hcat()</code>
[A, B, C, ...]	<code>vcate()</code>
[A B; C D; ...]	<code>hvcate()</code>
<code>A'</code>	<code>ctranspose()</code>
<code>A.'</code>	<code>transpose()</code>
<code>1:n</code>	<code>colon()</code>
<code>A[i]</code>	<code>getindex()</code>
<code>A[i]=x</code>	<code>setindex!()</code>

These functions are included in the `Base.Operators` module even though they do not have operator-like names.

1.8.5 Anonymous Functions

Functions in Julia are [first-class objects](#): they can be assigned to variables, 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:

```
julia> x -> x^2 + 2x - 1
(anonymous function)
```

This creates an unnamed function taking one argument `x` and returning the value of the polynomial $x^2 + 2x - 1$ at that value. 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 one wants 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()`.

1.8.6 Multiple Return Values

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and deconstructed 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> function foo(a,b)
    a+b, a*b
end;
```

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

```
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> x, y = foo(2,3);

julia> x
5

julia> y
6
```

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

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

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

1.8.7 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> 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> 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`.

On the flip side, it is often handy to “splice” 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 spliced 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 spliced into need not be a varargs function (although it often is):

```
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...)
no method baz{Int64,Int64,Int64}
```

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

1.8.8 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 library function `parseint(num, base)` interprets a string as a number in some base. The `base` argument defaults to 10. This behavior can be expressed concisely as:

```
function parseint(num, base=10)
    ###
end
```

With this definition, the function can be called with either one or two arguments, and 10 is automatically passed when a second argument is not specified:

```
julia> parseint("12", 10)
12

julia> parseint("12", 3)
5

julia> parseint("12")
12
```

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see [Methods](#)).

1.8.9 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
```

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

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

Inside `f`, `args` will be a collection of `(key, value)` tuples, where each `key` is a symbol. Such collections can be passed as keyword arguments using a semicolon in a call, e.g. `f(x, z=1; args...)`. Dictionaries can be used for this purpose.

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.

1.8.10 Evaluation Scope of Default Values

Optional and keyword arguments differ slightly in how their default values are evaluated. When optional argument default expressions are evaluated, only *previous* arguments are in scope. In contrast, *all* the arguments are in scope when keyword arguments default expressions are evaluated. 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`. However, if `a` and `b` were keyword arguments instead, then both would be created in the same scope and the `b` in `a=b` would refer to the subsequent argument `b` (shadowing any `b` in an outer scope), which would result in an undefined variable error (since the default expressions are evaluated left-to-right, and `b` has not been assigned yet).

1.8.11 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
```

In contrast to the `map()` example, here `io` is initialized by the *result* of `open("outfile", "w")`. The stream is then passed to your anonymous function, which performs the writing; finally, the `open()` function ensures that the stream is closed after your function exits. 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.

1.8.12 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](#).

1.9 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. *Tasks* 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.

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

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

1.9.2 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:

```
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> 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> 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, " than y.")
end;
```

`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> 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> if 1
    println("true")
end
ERROR: type: 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`.

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> 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> 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"
```

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



```
julia> factorial(5)
120

julia> factorial(0)
1

julia> factorial(-1)
ERROR: n must be non-negative
  in factorial at none:2
```

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> 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> 1 && true
ERROR: type: 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> true && (x = rand(2,2))
2x2 Array{Float64,2}:
 0.768448  0.673959
 0.940515  0.395453

julia> false && (x = rand(2,2))
false
```

1.9.4 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> i = 1;

julia> while i <= 5
    println(i)
    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 an other scope, in the `for` loop form, it is visible only inside of the `for` loop, and not 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: 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` 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 in ["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> while true
    println(i)
    if i >= 5
        break
    end
    i += 1
end

1
2
3
4
5

julia> for i = 1:1000
    println(i)
    if i >= 5
        break
    end
end

1
2
3
4
5
```

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 the `break` keyword.

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

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

A `break` statement inside such a loop exits the entire nest of loops, not just the inner one.

1.9.5 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, printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances, 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.

Exception
<code>ArgumentError</code>
<code>BoundsError</code>
<code>DivideError</code>
<code>DomainError</code>
<code>EOFError</code>
<code>ErrorException</code>
<code>InexactError</code>
<code>InterruptException</code>
<code>KeyError</code>
<code>LoadError</code>
<code>MemoryError</code>
<code>MethodError</code>
<code>OverflowError</code>
<code>ParseError</code>
<code>SystemError</code>
<code>TypeError</code>
<code>UndefRefError</code>
<code>UndefVarError</code>

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

```
julia> sqrt(-1)
ERROR: DomainError
sqrt will only return a complex result if called with a complex argument.
try sqrt(complex(x))
in sqrt at math.jl:131
```

You may define your own exceptions in the following way:

```
julia> type 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> f(x) = x>=0 ? exp(-x) : throw(DomainError())
f (generic function with 1 method)

julia> f(1)
0.36787944117144233
```

```
julia> f(-1)
ERROR: DomainError
  in f 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> typeof(DomainError()) <: Exception
true
```

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

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

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

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

```
julia> type MyUndefVarError <: Exception
    var::Symbol
end
julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined");
```

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
  in fussy_sqrt at none:1
```

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
  in verbose_fussy_sqrt at none:3
```

Warnings and informational messages

Julia also provides other functions that write messages to the standard error I/O, but do not throw any `Exceptions` and hence do not interrupt execution.:

```
julia> info("Hi"); 1+1
INFO: Hi
2

julia> warn("Hi"); 1+1
WARNING: Hi
2

julia> error("Hi"); 1+1
ERROR: Hi
  in error at error.jl:21
```

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. In this contrived example, the following 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
  in sqrt_second at none:7
```

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 `catch` clause is not strictly necessary; when omitted, the default return value is `false`. Note that this behavior will change in Julia version 0.4, where the return value will instead be `nothing`.

```
julia> try error() end
false
```

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.

1.9.6 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 the functions `produce()` and `consume()` for solving this problem. A producer is a function that calls `produce()` on each value it needs to produce:

```
julia> function producer()
    produce("start")
    for n=1:4
        produce(2n)
    end
    produce("stop")
end;
```

To consume values, first the producer is wrapped in a `Task`, then `consume()` is called repeatedly on that object:

```
julia> p = Task(producer);

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
6

julia> consume(p)
8

julia> consume(p)
"stop"
```

One way to think of this behavior is that `producer` was able to return multiple times. Between calls to `produce()`, the producer’s execution is suspended and the consumer has control.

A `Task` can be used as an iterable object in a `for` loop, in which case the loop variable takes on all the produced values:

```
julia> for x in Task(producer)
    println(x)
end
start
2
4
6
8
stop
```

Note that the `Task()` constructor expects a 0-argument function. A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0-argument *anonymous function*. This can be done either directly or by use of a convenience macro:

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

`produce()` and `consume()` do not launch threads that can run on separate CPUs. True kernel threads are discussed under the topic of *Parallel Computing*.

Core task operations

While `produce()` and `consume()` illustrate the essential nature of tasks, they are actually implemented as library functions using a more primitive function, `yieldto()`. `yieldto(task, value)` suspends the current task, switches to the specified `task`, and causes that task's last `yieldto()` call to return the specified `value`. Notice that `yieldto()` is the only operation required to use task-style control flow; instead of calling and returning we are always just switching to a different task. This is why this feature is also called “symmetric coroutines”; each task is switched to and from using the same mechanism.

`yieldto()` is powerful, but most uses of tasks do not invoke it directly. Consider why this might be. If you switch away from the current task, you will probably want to switch back to it at some point, but knowing when to switch back, and knowing which task has the responsibility of switching back, can require considerable coordination. For example, `produce()` needs to maintain some state to remember who the consumer is. Not needing to manually keep track of the consuming task is what makes `produce()` easier to use than `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 the standard library. 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 `@schedule` or `@async` macros (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
<code>:runnable</code>	Currently running, or available to be switched to
<code>:waiting</code>	Blocked waiting for a specific event
<code>:queued</code>	In the scheduler's run queue about to be restarted
<code>:done</code>	Successfully finished executing
<code>:failed</code>	Finished with an uncaught exception

1.10 Scope of Variables

The *scope* of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called `x` without the two `x`'s referring to the same thing. Similarly there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce *scope blocks*, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. The constructs introducing such blocks are:

- function bodies (*either syntax*)
- `while` loops
- `for` loops
- `try` blocks
- `catch` blocks
- `let` blocks
- `type` blocks.

Notably missing from this list are *begin blocks* and *if blocks*, which do *not* introduce new scope blocks.

Certain constructs introduce new variables into the current innermost scope. When a variable is introduced into a scope, it is also inherited by all inner scopes unless one of those inner scopes explicitly overrides it.

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` is found in the global scope (and if no global variable `x` existed, an undefined variable error would be raised):

```
function foo()
    x
end

function bar()
    x = 1
    foo()
end

x = 2

julia> bar()
2
```

If `foo` is instead defined inside `bar`, then it accesses the local `x` present in that function:

```
function bar()
    function foo()
        x
    end
    x = 1
    foo()
end

x = 2

julia> bar()
1
```

The constructs that introduce new variables into the current scope are as follows:

- A declaration `local x` or `const x` introduces a new local variable.
- A declaration `global x` makes `x` in the current scope and inner scopes refer to the global variable of that name.
- A function’s arguments are introduced as new local variables into the function’s body scope.
- An assignment `x = y` introduces a new local variable `x` only if `x` is neither declared global nor introduced as local by any enclosing scope before *or after* the current line of code.

In the following example, there is only one `x` assigned both inside and outside the `for` loop:

```
function foo(n)
    x = 0
    for i = 1:n
        x = x + 1
    end
    x
end

julia> foo(10)
10
```

In the next example, the loop has a separate `x` and the function always returns zero:

```
function foo(n)
    x = 0
    for i = 1:n
        local x
        x = i
    end
    x
end
```

```
julia> foo(10)
0
```

In this example, an `x` exists only inside the loop, and the function encounters an undefined variable error on its last line (unless there is a global variable `x`):

```
function foo(n)
    for i = 1:n
        x = i
    end
    x
end
```

```
julia> foo(10)
in foo: x not defined
```

A variable that is not assigned to or otherwise introduced locally defaults to global, so this function would return the value of the global `x` if there were such a variable, or produce an error if no such global existed. As a consequence, the only way to assign to a global variable inside a non-top-level scope is to explicitly declare the variable as global within some scope, since otherwise the assignment would introduce a new local rather than assigning to the global. This rule works out well in practice, since the vast majority of variables assigned inside functions are intended to be local variables, and using global variables should be the exception rather than the rule, and assigning new values to them even more so.

One last example shows that an outer assignment introducing `x` need not come before an inner usage:

```
function foo(n)
    f = y -> n + x + y
    x = 1
    f(2)
end
```

```
julia> foo(10)
13
```

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, both of which one typically sees in C programs. As an example, here is an inefficient, mutually recursive way to test if positive integers are even or odd:

```
even(n) = n == 0 ? true  : odd(n-1)
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 taken as examples.

Since functions can be used before they are defined, as long as they are defined by the time they are actually called, no syntax for forward declarations is necessary, and definitions can be ordered arbitrarily.

At the interactive prompt, variable scope works the same way as anywhere else. The prompt behaves as if there is scope block wrapped around everything you type, except that this scope block is identified with the global scope. This is especially evident in the case of assignments:

```
julia> for i = 1:1; y = 10; end
```

```
julia> y
ERROR: y not defined
```

```
julia> y = 0
0
```

```
julia> for i = 1:1; y = 10; end
```

```
julia> y
10
```

In the former case, `y` only exists inside of the `for` loop. In the latter case, an outer `y` has been introduced and so is inherited within the loop. Due to the special identification of the prompt's scope block with the global scope, it is not necessary to declare `global y` inside the loop. However, in code not entered into the interactive prompt this declaration would be necessary in order to modify a global variable.

Multiple variables can be declared global using the following syntax:

```
function foo()
    global x=1, y="bar", z=3
end
```

```
julia> foo()
3
```

```
julia> x
1
```

```
julia> y
"bar"
```

```
julia> z
3
```

The `let` statement provides a different way to introduce variables. 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:

```
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. Here is an example where the behavior of `let` is needed:

```
Fs = cell(2)
i = 1
while i <= 2
    Fs[i] = ()->i
    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`:

```
Fs = cell(2)
i = 1
while i <= 2
    let i = i
        Fs[i] = ()->i
    end
    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> begin
    local x = 1
    begin
        local x = 2
    end
    x
end
```

ERROR: syntax: `local` "x" declared twice

```
julia> begin
    local x = 1
    let
        local x = 2
    end
    x
end
1
```

The first example is invalid because you cannot declare the same variable as `local` in the same scope twice. The second example is valid since the `let` introduces a new scope block, so the inner `local x` is a different variable than the outer `local x`.

1.10.1 For Loops and Comprehensions

`for` loops and *comprehensions* have a special additional behavior: any new variables introduced in their body scopes are freshly allocated for each loop iteration. Therefore these constructs are similar to `while` loops with `let` blocks inside:

```
Fs = cell(2)
for i = 1:2
    Fs[i] = ()->i
end
```

```
julia> Fs[1]()
1
```

```
julia> Fs[2]()
2
```

`for` loops will reuse existing variables for iteration:

```
i = 0
for i = 1:3
end
i # here equal to 3
```

However, comprehensions do not do this, and always freshly allocate their iteration variables:

```
x = 0
[ x for x=1:3 ]
x # here still equal to 0
```

1.10.2 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:

```
const e = 2.71828182845904523536
const pi = 3.14159265358979323846
```

The `const` declaration is allowed on both global and local variables, but is especially useful for 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 for performance purposes.

Special top-level assignments, such as those performed by the `function` and `type` 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.

1.11 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 programs without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously “untyped” code. Doing so will typically increase both the performance and robustness of these systems, and perhaps somewhat counterintuitively, often significantly simplify them.

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.

1.11.1 Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect,
2. To provide extra type information to the compiler, which can then improve performance in some cases

When appended to an expression computing a *value*, the `::` operator is read as “is an instance of”. It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation — recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:


```
julia> (1+2)::FloatingPoint
ERROR: type: typeassert: expected FloatingPoint, got Int64

julia> (1+2)::Int
3
```

This allows a type assertion to be attached to any expression in-place. The most common usage of `::` as an assertion is in function/methods signatures, such as `f(x::Int8) = ...` (see [Methods](#)).

When appended to a *variable* in a statement context, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using `convert()`:

```
julia> function foo()
           x::Int8 = 1000
           x
       end
foo (generic function with 1 method)

julia> foo()
-24

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.

The “declaration” behavior only occurs in specific contexts:

```
x::Int8           # a variable by itself
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. Note that in a function return statement, the first two of the above expressions compute a value and then `::` is a type assertion and not a declaration.

1.11.2 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` keyword. The general syntaxes for declaring an abstract type are:

```
abstract <name>
abstract <name> <: <supertype>
```

The `abstract` keyword introduces a new abstract type, whose name is given by `<name>`. This name can be optionally followed by `<: <supertype>` 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 called `Nothing`. It is the exact opposite of `Any`: no object is an instance of `Nothing` and all types are supertypes of `Nothing`.

Let’s consider some of the abstract types that make up Julia’s numerical hierarchy:

```
abstract Number
abstract Real <: Number
abstract FloatingPoint <: Real
abstract Integer <: Real
abstract Signed <: Integer
abstract Unsigned <: Integer
```

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 `FloatingPoint`, 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, `FloatingPoint` is a proper subtype of `Real`, including only floating-point representations of real numbers. Integers are further subdivided into `Signed` and `Unsigned` varieties.

The `<: <supertype>` operator in general means “is a subtype of”, and, used in declarations like this, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: FloatingPoint
false
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
function myplus(x,y)
    x+y
end
```

The first thing to note is that the above argument declarations are equivalent to `x::Any` and `y::Any`. When this function is invoked, say as `myplus(2,5)`, the dispatcher chooses the most specific method named `myplus` that matches the given arguments. (See [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:

```
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](#).)

1.11.3 Bits Types

A bits type is a concrete type whose data consists of plain old bits. Classic examples of bits types are integers and floating-point values. Unlike most languages, Julia lets you declare your own bits types, rather than providing only a fixed set of built-in bits types. In fact, the standard bits types are all defined in the language itself:

```
bitstype 16 Float16 <: FloatingPoint
bitstype 32 Float32 <: FloatingPoint
bitstype 64 Float64 <: FloatingPoint

bitstype 8 Bool <: Integer
bitstype 32 Char <: Integer

bitstype 8 Int8 <: Signed
bitstype 8 UInt8 <: Unsigned
bitstype 16 Int16 <: Signed
bitstype 16 UInt16 <: Unsigned
bitstype 32 Int32 <: Signed
bitstype 32 UInt32 <: Unsigned
bitstype 64 Int64 <: Signed
bitstype 64 UInt64 <: Unsigned
bitstype 128 Int128 <: Signed
bitstype 128 UInt128 <: Unsigned
```

The general syntaxes for declaration of a `bitstype` are:

```
bitstype <bits> <name>
bitstype <bits> <name> <: <supertype>
```

The number of bits indicates how much storage the type requires and the name gives the new type a name. A bits 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. Another 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`.

1.11.4 Composite Types

Composite types are called records, structures (`structs` in C), 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 Python and Ruby, 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.

Since composite types are the most common form of user-defined concrete type, they are simply introduced with the `type` keyword followed by a block of field names, optionally annotated with types using the `::` operator:

```
julia> type 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 composite type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo (constructor with 2 methods)
```

When a type is applied like a function it is called a *constructor*. Two constructors are generated automatically (these are called *default constructors*). One accepts any arguments and calls `convert()` to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Since the `bar` field is unconstrained in type, any value will do. However, the value for `baz` must be convertible to `Int`:

```
julia> Foo(), 23.5, 1)
ERROR: InexactError()
  in Foo at no file
```

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

```
julia> names(foo)
3-element Array{Symbol,1}:
 :bar
 :baz
 :qux
```

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

```
julia> foo.bar
"Hello, world."

julia> foo.baz
```

```
23
```

```
julia> foo.qux
1.5
```

You can also change the values as one would expect:

```
julia> foo.qux = 2
2.0
```

```
julia> foo.bar = 1//2
1//2
```

Composite types with no fields are singletons; there can be only one instance of such types:

```
type NoFields
end
```

```
julia> is(NoFields(), NoFields())
true
```

The `is` 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](#).

1.11.5 Immutable Composite Types

It is also possible to define *immutable* composite types by using the keyword `immutable` instead of `type`:

```
immutable Complex
    real::Float64
    imag::Float64
end
```

Such types behave much like other composite types, except that instances of them cannot be modified. Immutable types have several advantages:

- They are more efficient in some cases. Types like the `Complex` example above 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.

A useful way to think about immutable composites is that each instance is associated with specific field values — the field values alone tell you everything about the object. In contrast, a mutable object is like a little container that might hold different values over time, and so is not identified with specific field values. In deciding whether to make a type immutable, 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:

- An object with an immutable type is passed around (both in assignment statements and in function calls) by copying, whereas a mutable type is passed around by reference.
- It is not permitted to modify the fields of a composite immutable type.

It is instructive, particularly for readers whose background is C/C++, to consider why these two properties go hand in hand. If they were separated, i.e., if the fields of objects passed around by copying could be modified, then it would become more difficult to reason about certain instances of generic code. For example, suppose `x` is a function argument of an abstract type, and suppose that the function changes a field: `x.isprocessed = true`. Depending on whether `x` is passed by copying or by reference, this statement may or may not alter the actual argument in the calling routine. Julia sidesteps the possibility of creating functions with unknown effects in this scenario by forbidding modification of fields of objects passed around by copying.

1.11.6 Declared Types

The three kinds of types discussed in the previous three sections are actually all closely related. They share the same key properties:

- They are explicitly declared.
- They have names.
- They have explicitly declared supertypes.
- They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept, `DataType`, which is the type of any of these types:

```
julia> typeof(Real)
DataType
```

```
julia> typeof(Int)
DataType
```

A `DataType` may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a bits 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 either an instance of some `DataType`, or is a tuple.

1.11.7 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. The type of a tuple of values is the tuple of types of values:

```
julia> typeof((1, "foo", 2.5))
(Int64, ASCIIString, Float64)
```

Accordingly, a tuple of types can be used anywhere a type is expected:

```
julia> (1, "foo", 2.5) :: (Int64, String, Any)
(1, "foo", 2.5)
```

```
julia> (1, "foo", 2.5) :: (Int64, String, Float32)
ERROR: type: typeassert: expected (Int64, String, Float32), got (Int64, ASCIIString, Float64)
```

If one of the components of the tuple is not a type, however, you will get an error:

```
julia> (1, "foo", 2.5) :: (Int64, String, 3)
ERROR: type: typeassert: expected Type{T<:Top}, got (DataType, DataType, Int64)
```

Note that the empty tuple `()` is its own type:

```
julia> typeof(())
()
```

Tuple types are *covariant* in their constituent types, which means that one tuple type is a subtype of another if elements of the first are subtypes of the corresponding elements of the second. For example:

```
julia> (Int, String) <: (Real, Any)
true
```

```
julia> (Int, String) <: (Real, Real)
false
```

```
julia> (Int, String) <: (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).

1.11.8 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` function:

```
julia> IntOrString = Union{Int, String}
Union{String, Int64}
```

```
julia> 1 :: IntOrString
1
```

```
julia> "Hello!" :: IntOrString
"Hello!"
```

```
julia> 1.0 :: IntOrString
ERROR: type: typeassert: expected Union{String, Int64}, got Float64
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer. The union of no types is the “bottom” type, `Nothing`:

```
julia> Union{ }
Nothing
```

Recall from the discussion above that `Nothing` is the abstract type which is the subtype of all other types, and which no object is an instance of. Since a zero-argument `Union` call has no argument types for objects to be instances of, it should produce a type which no objects are instances of — i.e. `Nothing`.

1.11.9 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 bits types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
type 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 an integer, 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{String}`, `Point{Int64}`, etc. Each of these is now a usable concrete type:

```
julia> Point{Float64}
Point{Float64} (constructor with 1 method)

julia> Point{String}
Point{String} (constructor with 1 method)
```

The type `Point{Float64}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{String}` is a “point” whose “coordinates” are string objects (see [Strings](#)). However, `Point` itself is also a valid type object:

```
julia> Point
Point{T} (constructor with 1 method)
```

Here the `T` is the dummy type symbol used in the original declaration of `Point`. What does `Point` by itself mean? It is an abstract type that contains all the specific instances `Point{Float64}`, `Point{String}`, etc.:

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

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

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

```
julia> Float64 <: Point
false

julia> String <: Point
false
```

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

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```


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.

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} (constructor with 1 method)
```

For the default constructor, exactly one argument must be supplied for each field:

```
julia> Point{Float64}(1.0)
ERROR: `Point{Float64}` has no method matching Point{Float64}(::Float64)

julia> Point{Float64}(1.0,2.0,3.0)
ERROR: `Point{Float64}` has no method matching Point{Float64}(::Float64, ::Float64, ::Float64)
```

Only one default constructor is generated for parametric types, since overriding it is not possible. This constructor accepts any arguments and converts them to the field types.

In many cases, it is redundant to provide the type of `Point` object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply `Point` itself as a constructor, provided that the implied value of the parameter type `T` is unambiguous:

```
julia> Point(1.0,2.0)
Point{Float64}(1.0,2.0)

julia> typeof(ans)
Point{Float64} (constructor with 1 method)

julia> Point(1,2)
Point{Int64}(1,2)

julia> typeof(ans)
Point{Int64} (constructor with 1 method)
```

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 no method error:

```
julia> Point(1,2.5)
ERROR: 'Point{T}' has no method matching Point{T}(::Int64, ::Float64)
```

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:

```
abstract Pointy{T}
```

With this declaration, `Pointy{T}` is a distinct abstract type for each type or integer value of `T`. As with parametric composite types, each such instance is a subtype of `Pointy`:

```
julia> Pointy{Int64} <: Pointy
true
```

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

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

```
julia> Pointy{Float64} <: Pointy{Real}
false
```

```
julia> Pointy{Real} <: Pointy{Float64}
false
```

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:

```
type Point{T} <: Pointy{T}
    x::T
    y::T
end
```

Given such a declaration, for each choice of `T`, we have `Point{T}` as a subtype of `Pointy{T}`:

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

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

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

This relationship is also invariant:

```
julia> Point{Float64} <: Pointy{Real}
false
```

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 $x = y$:

```

type 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:

```

abstract Pointy{T<:Real}

```

With such a declaration, it is acceptable to use any type that is a subtype of `Real` in place of `T`, but not types that are not subtypes of `Real`:

```

julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{String}
ERROR: type: Pointy: in T, expected T<:Real, got Type{String}

julia> Pointy{1}
ERROR: type: Pointy: in T, expected T<:Real, got Int64

```

Type parameters for parametric composite types can be restricted in the same manner:

```

type 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:

```

immutable 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.

Singleton Types

There is a special kind of abstract parametric type that must be mentioned here: singleton types. For each type, `T`, the “singleton type” `Type{T}` is an abstract type whose only instance is the object `T`. Since the definition is a little difficult to parse, let’s look at some examples:

```

julia> isa(Float64, Type{Float64})
true

```

```
julia> isa(Real, Type{Float64})
false
```

```
julia> isa(Real, Type{Real})
true
```

```
julia> isa(Float64, Type{Real})
false
```

In other words, `isa(A, Type{B})` is true if and only if A and B are the same object and that object is a type. Without the parameter, `Type` is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

```
julia> isa(Type{Float64}, Type)
true
```

```
julia> isa(Float64, Type)
true
```

```
julia> isa(Real, Type)
true
```

Any object that is not a type is not an instance of `Type`:

```
julia> isa(1, Type)
false
```

```
julia> isa("foo", Type)
false
```

Until we discuss *Parametric Methods* and *conversions*, it is difficult to explain the utility of the singleton type construct, but in short, it allows one to specialize function behavior on specific type *values*. This is useful for writing methods (especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

A few popular languages have singleton types, including Haskell, Scala and Ruby. In general usage, the term “singleton type” refers to a type whose only instance is a single value. This meaning applies to Julia’s singleton types, but with that caveat that only type objects have singleton types.

Parametric Bits Types

Bits types can also be declared parametrically. For example, pointers are represented as boxed bits types which would be declared in Julia like this:

```
# 32-bit system:
bitstype 32 Ptr{T}
```

```
# 64-bit system:
bitstype 64 Ptr{T}
```

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 subtype of the umbrella `Ptr` type:

```
julia> Ptr{Float64} <: Ptr
true
```

```
julia> Ptr{Int64} <: Ptr
true
```

1.11.10 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. For such occasions, Julia provides the `typealias` mechanism. For example, `UInt` is type aliased to either `UInt32` or `UInt64` as is appropriate for the size of pointers on the system:

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

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

This is accomplished via the following code in `base/boot.jl`:

```
if is(Int, Int64)
    typealias UInt UInt64
else
    typealias 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`.

For parametric types, `typealias` can be convenient for providing names for cases where some of the parameter choices are fixed. Julia’s arrays have type `Array{T,N}` where `T` is the element type and `N` is the number of array dimensions. For convenience, writing `Array{Float64}` allows one to specify the element type without specifying the dimension:

```
julia> Array{Float64,1} <: Array{Float64} <: Array
true
```

However, there is no way to equally simply restrict just the dimension but not the element type. Yet, one often needs to ensure an object is a vector or a matrix (imposing restrictions on the number of dimensions). For that reason, the following type aliases are provided:

```
typealias Vector{T} Array{T,1}
typealias Matrix{T} Array{T,2}
```

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 `Matrix` for the abstract type including all two-dimensional dense arrays of any element type.

This declaration of `Vector` creates a subtype relation `Vector{Int} <: Vector`. However, it is not always the case that a parametric `typealias` statement creates such a relation; for example, the statement:

```
typealias AA{T} Array{Array{T,1},1}
```

does not create the relation `AA{Int} <: AA`. The reason is that `Array{Array{T,1},1}` is not an abstract type at all; in fact, it is a concrete type describing a 1-dimensional array in which each entry is an object of type `Array{T,1}` for some value of `T`.

1.11.11 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, FloatingPoint)
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)
DataType
```

```
julia> typeof(Union{Real, Float64, Rational})
DataType
```

```
julia> typeof((Rational, None))
(DataType, UnionType)
```

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(UnionType)
DataType
```

The reader may note that `DataType` shares with the empty tuple (see [above](#)), the distinction of being its own type (i.e. a fixed point of the `typeof()` function). This leaves any number of tuple types recursively built with `()` and `DataType` as their only atomic values, which are their own type:

```
julia> typeof(())
()
```

```
julia> typeof(DataType)
DataType
```

```
julia> typeof(((),))
((),)
```

```
julia> typeof((DataType,))
(DataType,)
```

```
julia> typeof(((), DataType))
((), DataType)
```

All fixed points of `typeof()` are like this.

Another operation that applies to some types is `super()`, which reveals a type's supertype. Only declared types (`DataType`) have unambiguous supertypes:

```
julia> super(Float64)
FloatingPoint
```

```
julia> super(Number)
Any
```

```
julia> super(String)
Any
```

```
julia> super(Any)
Any
```

If you apply `super()` to other type objects (or non-type objects), a `MethodError` is raised:

```
julia> super(Union{Float64, Int64})
ERROR: `super` has no method matching super(::Type{Union{Float64, Int64}})
```

```
julia> super(None)
ERROR: `super` has no method matching super(::Type{None})
```

```
julia> super((Float64, Int64))
ERROR: `super` has no method matching super(::Type{(Float64, Int64)})
```

1.12 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.

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.

1.12.1 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 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;
```

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 “no method” error:

```
julia> f(2.0, 3)
ERROR: `f` has no method matching f(::Float64, ::Int64)

julia> f(float32(2.0), 3.0)
ERROR: `f` has no method matching f(::Float32, ::Float64)

julia> f(2.0, "3.0")
ERROR: `f` has no method matching f(::Float64, ::ASCIIString)

julia> f("2.0", "3.0")
ERROR: `f` has no method matching f(::ASCIIString, ::ASCIIString)
```

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;

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. [Clarke61]

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 “no method” error:

```
julia> f("foo", 3)
ERROR: `f` has no method matching f(::ASCIIString, ::Int64)
```

```
julia> f()
ERROR: `f` has no method matching f()
```

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":
f(x::Float64,y::Float64) at none:1
f(x::Number,y::Number) 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> f(x,y) = println("Whoa there, Nelly.");

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 is 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> methods(+)
# 125 methods for generic function "+":
+(x::Bool) at bool.jl:36
+(x::Bool,y::Bool) at bool.jl:39
+(y::FloatingPoint,x::Bool) at bool.jl:49
```

```
+ (A::BitArray{N}, B::BitArray{N}) at bitarray.jl:848
+ (A::Union{DenseArray{Bool,N}, SubArray{Bool,N,A<:DenseArray{T,N}, I<: (Union{Range{Int64}, Int64}...),)}}
+ {S,T} (A::Union{SubArray{S,N,A<:DenseArray{T,N}, I<: (Union{Range{Int64}, Int64}...),}}, DenseArray{S,N})
+ {T<:Union{Int16, Int32, Int8}} (x::T<:Union{Int16, Int32, Int8}, y::T<:Union{Int16, Int32, Int8}) at int.jl:33
+ {T<:Union{UInt32, UInt16, UInt8}} (x::T<:Union{UInt32, UInt16, UInt8}, y::T<:Union{UInt32, UInt16, UInt8}) at
+ (x::Int64, y::Int64) at int.jl:33
+ (x::UInt64, y::UInt64) at int.jl:34
+ (x::Int128, y::Int128) at int.jl:35
+ (x::UInt128, y::UInt128) at int.jl:36
+ (x::Float32, y::Float32) at float.jl:119
+ (x::Float64, y::Float64) at float.jl:120
+ (z::Complex{T<:Real}, w::Complex{T<:Real}) at complex.jl:110
+ (x::Real, z::Complex{T<:Real}) at complex.jl:120
+ (z::Complex{T<:Real}, x::Real) at complex.jl:121
+ (x::Rational{T<:Integer}, y::Rational{T<:Integer}) at rational.jl:113
+ (x::Char, y::Char) at char.jl:23
+ (x::Char, y::Integer) at char.jl:26
+ (x::Integer, y::Char) at char.jl:27
+ (a::Float16, b::Float16) at float16.jl:132
+ (x::BigInt, y::BigInt) at gmp.jl:194
+ (a::BigInt, b::BigInt, c::BigInt) at gmp.jl:217
+ (a::BigInt, b::BigInt, c::BigInt, d::BigInt) at gmp.jl:223
+ (a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) at gmp.jl:230
+ (x::BigInt, c::UInt64) at gmp.jl:242
+ (c::UInt64, x::BigInt) at gmp.jl:246
+ (c::Union{UInt32, UInt16, UInt8, UInt64}, x::BigInt) at gmp.jl:247
+ (x::BigInt, c::Union{UInt32, UInt16, UInt8, UInt64}) at gmp.jl:248
+ (x::BigInt, c::Union{Int64, Int16, Int32, Int8}) at gmp.jl:249
+ (c::Union{Int64, Int16, Int32, Int8}, x::BigInt) at gmp.jl:250
+ (x::BigFloat, c::UInt64) at mpfr.jl:147
+ (c::UInt64, x::BigFloat) at mpfr.jl:151
+ (c::Union{UInt32, UInt16, UInt8, UInt64}, x::BigFloat) at mpfr.jl:152
+ (x::BigFloat, c::Union{UInt32, UInt16, UInt8, UInt64}) at mpfr.jl:153
+ (x::BigFloat, c::Int64) at mpfr.jl:157
+ (c::Int64, x::BigFloat) at mpfr.jl:161
+ (x::BigFloat, c::Union{Int64, Int16, Int32, Int8}) at mpfr.jl:162
+ (c::Union{Int64, Int16, Int32, Int8}, x::BigFloat) at mpfr.jl:163
+ (x::BigFloat, c::Float64) at mpfr.jl:167
+ (c::Float64, x::BigFloat) at mpfr.jl:171
+ (c::Float32, x::BigFloat) at mpfr.jl:172
+ (x::BigFloat, c::Float32) at mpfr.jl:173
+ (x::BigFloat, c::BigInt) at mpfr.jl:177
+ (c::BigInt, x::BigFloat) at mpfr.jl:181
+ (x::BigFloat, y::BigFloat) at mpfr.jl:328
+ (a::BigFloat, b::BigFloat, c::BigFloat) at mpfr.jl:339
+ (a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) at mpfr.jl:345
+ (a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat, e::BigFloat) at mpfr.jl:352
+ (x::MathConst{sym}, y::MathConst{sym}) at constants.jl:23
+ {T<:Number} (x::T<:Number, y::T<:Number) at promotion.jl:188
+ {T<:FloatingPoint} (x::Bool, y::T<:FloatingPoint) at bool.jl:46
+ (x::Number, y::Number) at promotion.jl:158
+ (x::Integer, y::Ptr{T}) at pointer.jl:68
+ (x::Bool, A::AbstractArray{Bool,N}) at array.jl:767
+ (x::Number) at operators.jl:71
+ (r1::OrdinalRange{T,S}, r2::OrdinalRange{T,S}) at operators.jl:325
+ {T<:FloatingPoint} (r1::FloatRange{T<:FloatingPoint}, r2::FloatRange{T<:FloatingPoint}) at operators.jl:348
+ (r1::FloatRange{T<:FloatingPoint}, r2::FloatRange{T<:FloatingPoint}) at operators.jl:348
+ (r1::FloatRange{T<:FloatingPoint}, r2::OrdinalRange{T,S}) at operators.jl:349
```

```

+(r1::OrdinalRange{T,S},r2::FloatRange{T<:FloatingPoint}) at operators.jl:350
+(x::Ptr{T},y::Integer) at pointer.jl:66
+{S,T<:Real} (A::Union{SubArray{S,N,A<:DenseArray{T,N}},I<:(Union{Range{Int64},Int64}...)},DenseArray{
+{S<:Real,T} (A::Range{S<:Real},B::Union{SubArray{T,N,A<:DenseArray{T,N}},I<:(Union{Range{Int64},Int64}...)}
+(A::AbstractArray{Bool,N},x::Bool) at array.jl:766
+{Tv,Ti} (A::SparseMatrixCSC{Tv,Ti},B::SparseMatrixCSC{Tv,Ti}) at sparse/sparsematrix.jl:530
+{TvA,TiA,TvB,TiB} (A::SparseMatrixCSC{TvA,TiA},B::SparseMatrixCSC{TvB,TiB}) at sparse/sparsematrix.jl:530
+(A::SparseMatrixCSC{Tv,Ti<:Integer},B::Array{T,N}) at sparse/sparsematrix.jl:621
+(A::Array{T,N},B::SparseMatrixCSC{Tv,Ti<:Integer}) at sparse/sparsematrix.jl:623
+(A::SymTridiagonal{T},B::SymTridiagonal{T}) at linalg/tridiag.jl:45
+(A::Tridiagonal{T},B::Tridiagonal{T}) at linalg/tridiag.jl:207
+(A::Tridiagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:99
+(A::SymTridiagonal{T},B::Tridiagonal{T}) at linalg/special.jl:98
+{T,MT,uplo} (A::Triangular{T,MT,uplo,IsUnit},B::Triangular{T,MT,uplo,IsUnit}) at linalg/triangular.jl:44
+{T,MT,uplo1,uplo2} (A::Triangular{T,MT,uplo1,IsUnit},B::Triangular{T,MT,uplo2,IsUnit}) at linalg/triangular.jl:44
+(Da::Diagonal{T},Db::Diagonal{T}) at linalg/diagonal.jl:44
+(A::Bidiagonal{T},B::Bidiagonal{T}) at linalg/bidiag.jl:92
+{T} (B::BitArray{2},J::UniformScaling{T}) at linalg/uniformscaling.jl:26
+(A::Diagonal{T},B::Bidiagonal{T}) at linalg/special.jl:89
+(A::Bidiagonal{T},B::Diagonal{T}) at linalg/special.jl:90
+(A::Diagonal{T},B::Tridiagonal{T}) at linalg/special.jl:89
+(A::Tridiagonal{T},B::Diagonal{T}) at linalg/special.jl:90
+(A::Diagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Diagonal{T}) at linalg/special.jl:90
+(A::Diagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Diagonal{T}) at linalg/special.jl:90
+(A::Bidiagonal{T},B::Tridiagonal{T}) at linalg/special.jl:89
+(A::Tridiagonal{T},B::Bidiagonal{T}) at linalg/special.jl:90
+(A::Bidiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Bidiagonal{T}) at linalg/special.jl:90
+(A::Bidiagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Bidiagonal{T}) at linalg/special.jl:90
+(A::Tridiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Tridiagonal{T}) at linalg/special.jl:90
+(A::Tridiagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Tridiagonal{T}) at linalg/special.jl:90
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.jl:90
+(A::SymTridiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.jl:98
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::SymTridiagonal{T}) at linalg/special.jl:99
+(A::SymTridiagonal{T},B::Array{T,2}) at linalg/special.jl:98
+(A::Array{T,2},B::SymTridiagonal{T}) at linalg/special.jl:99
+(A::Diagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:107
+(A::SymTridiagonal{T},B::Diagonal{T}) at linalg/special.jl:108
+(A::Bidiagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:107
+(A::SymTridiagonal{T},B::Bidiagonal{T}) at linalg/special.jl:108
+{T<:Number} (x::AbstractArray{T<:Number,N}) at abstractarray.jl:358
+(A::AbstractArray{T,N},x::Number) at array.jl:770
+(x::Number,A::AbstractArray{T,N}) at array.jl:771
+(J1::UniformScaling{T<:Number},J2::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:25
+(J::UniformScaling{T<:Number},B::BitArray{2}) at linalg/uniformscaling.jl:27
+(J::UniformScaling{T<:Number},A::AbstractArray{T,2}) at linalg/uniformscaling.jl:28
+(J::UniformScaling{T<:Number},x::Number) at linalg/uniformscaling.jl:29
+(x::Number,J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:30
+{TA,TJ} (A::AbstractArray{TA,2},J::UniformScaling{TJ}) at linalg/uniformscaling.jl:33
+{T} (a::HierarchicalValue{T},b::HierarchicalValue{T}) at pkg/resolve/versionweight.jl:19
+(a::VWPreBuildItem,b::VWPreBuildItem) at pkg/resolve/versionweight.jl:82
+(a::VWPreBuild,b::VWPreBuild) at pkg/resolve/versionweight.jl:120

```

```
+ (a::VersionWeight,b::VersionWeight) at pkg/resolve/versionweight.jl:164
+ (a::FieldValue,b::FieldValue) at pkg/resolve/fieldvalue.jl:41
+ (a::Vec2,b::Vec2) at graphics.jl:60
+ (bb1::BoundingBox,bb2::BoundingBox) at graphics.jl:123
+ (a,b,c) at operators.jl:82
+ (a,b,c,xs...) at operators.jl:83
```

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.

1.12.2 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> g(x::Float64, y) = 2x + y;

julia> g(x, y::Float64) = x + 2y;
Warning: New definition
      g(Any,Float64) at none:1
is ambiguous with:
      g(Float64,Any) at none:1.
To fix, define
      g(Float64,Float64)
before the new definition.
```

```
julia> g(2.0, 3)
7.0
```

```
julia> g(2, 3.0)
8.0
```

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

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 warns you about this ambiguity, but allows you to proceed, arbitrarily picking a method. You should avoid method ambiguities by specifying an appropriate method for the intersection case:

```
julia> g(x::Float64, y::Float64) = 2x + 2y;

julia> g(x::Float64, y) = 2x + y;

julia> g(x, y::Float64) = x + 2y;

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

To suppress Julia's warning, the disambiguating method must be defined first, since otherwise the ambiguity exists, if

transiently, until the more specific method is defined.

1.12.3 Parametric Methods

Method definitions can optionally have type parameters immediately after the method name and before the parameter tuple:

```
julia> same_type{T}(x::T, y::T) = true;

julia> same_type(x,y) = false;
```

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
```

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 parameters: 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{T}(v::Vector{T}, x::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: `myappend` has no method matching myappend(::Array{Int64,1}, ::Float64)

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: `myappend` has no method matching myappend(::Array{Float64,1}, ::Int64)
```

As you can see, the type of the appended element must match the element type of the vector it is appended to, or a “no method” error is raised. In the following example, the method type parameter `T` is used as the return value:

```
julia> mytypeof{T}(x::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:

```
same_type_numeric{T<:Number}(x::T, y::T) = true
same_type_numeric(x::Number, y::Number) = false

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)
no method same_type_numeric{ASCIIString,Float64}

julia> same_type_numeric("foo", "bar")
no method same_type_numeric{ASCIIString,ASCIIString}

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.

1.12.4 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)
```

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.

1.13 Constructors

Constructors² 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:

```
type 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.

1.13.1 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:

```
Foo(x) = Foo(x,x)

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:

```
Foo() = Foo(0)

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

² 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.

methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

1.13.2 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:

```
type 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 $x \leq y$:

```
julia> OrderedPair(1,2)
OrderedPair{Real,Real}(1,2)

julia> OrderedPair(2,1)
ERROR: out of order
in OrderedPair at none:5
```

You can still 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.

Of course, if the type is declared as `immutable`, then its constructor-provided invariants are fully enforced. This is an important consideration when deciding whether a type should be `immutable`.

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:

```
type 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:


```

type T1
    x::Int64
end

type 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.

1.13.3 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:

```

type 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:

```
b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, we take another crack at defining the `SelfReferential` type, with a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```

type 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> is(x, x)
true

julia> is(x, x.obj)
true

julia> is(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> type 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> z.xx
ERROR: 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 bits types (e.g. `Int`) and immutable structs of other plain data types. The initial contents of a plain data type is undefined:

```
julia> type 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:

```
type 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.

1.13.4 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> type Point{T<:Real}
      x::T
      y::T
end

## implicit T ##

julia> Point(1,2)
Point{Int64}(1,2)

julia> Point(1.0,2.5)
Point{Float64}(1.0,2.5)

julia> Point(1,2.5)
ERROR: `Point{T<:Real}` has no method matching Point{T<:Real}(::Int64, ::Float64)

## explicit T ##

julia> Point{Int64}(1,2)
Point{Int64}(1,2)

julia> Point{Int64}(1.0,2.5)
ERROR: InexactError()
  in Point at no file

julia> Point{Float64}(1.0,2.5)
Point{Float64}(1.0,2.5)

julia> Point{Float64}(1,2)
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:

```
type Point{T<:Real}
  x::T
  y::T

  Point(x,y) = new(x,y)
end

Point{T<:Real}(x::T, y::T) = Point{T}(x,y)
```

Some features of parametric constructor definitions at work here deserve comment. First, inner constructor declarations always define methods of `Point{T}` rather than methods of the general `Point` constructor function. Since `Point` is not a concrete type, it makes no sense for it to even have inner constructor methods at all. Thus, the inner method declaration `Point(x,y) = new(x,y)` provides an inner constructor method for each

value of `T`. It is this method declaration that defines the behavior of constructor calls with explicit type parameters like `Point{Int64}(1,2)` and `Point{Float64}(1.0,2.0)`. 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} (constructor with 1 method)
```

However, other similar calls still don’t work:

```
julia> Point(1.5,2)
ERROR: `Point{T<:Real}` has no method matching Point{T<:Real}(::Float64, ::Int64)
```

For a much more general way of making all such calls work sensibly, see [Conversion and Promotion](#). At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

The `promote` function converts all its arguments to a common type — in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point{Float64}(1.5,2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1,1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0,0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

1.13.5 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, here is beginning of `rational.jl`, which implements Julia’s *Rational Numbers*:

```
immutable Rational{T<:Integer} <: Real
    num::T
```

```

den::T

function Rational(num::T, den::T)
    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

Rational{T<:Integer}(n::T, d::T) = Rational{T}(n,d)
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
Rational(n::Integer) = Rational(n,one(n))

//(n::Integer, d::Integer) = Rational(n,d)
//(x::Rational, y::Integer) = x.num // (x.den*y)
//(x::Integer, y::Rational) = (x*y.den) // y.num
//(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
//(x::Real, y::Complex) = x*y'//real(y*y')

function //(x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end

```

The first line — `immutable Rational{T<:Int} <: Real` — declares that `Rational` 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 `Rational{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. `Rational` 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 `Rational`, we can be certain that `Rational` objects are always constructed in this normalized form.

`Rational` 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 have a number of methods for the `//` operator, which provides a syntax for writing rationals. 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 `Rational` by applying the `Rational` 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{Rational}` — a complex number whose real and imaginary parts are rationals:

```

julia> (1 + 2im)//(1 - 2im)
-3//5 + 4//5*im

```

```
julia> typeof(ans)
Complex{Rational{Int64}} (constructor with 1 method)

julia> ans <: Complex{Rational}
false
```

Thus, although the `//` operator usually returns an instance of `Rational`, if either of its arguments are complex integers, it will return an instance of `Complex{Rational}` instead. The interested reader should consider perusing the rest of `rational.jl`: it is short, self-contained, and implements an entire basic Julia type in just a little over a hundred lines of code.

1.14 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.

1.14.1 Conversion

Conversion of values to various types is performed by the `convert` function. The `convert` function generally takes two arguments: the first is a type object while the second is a value to convert to that type; the returned value is the

value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert{UInt8, x}
0x0c

julia> typeof(ans)
UInt8

julia> convert{Float64, x}
12.0

julia> typeof(ans)
Float64
```

Conversion isn't always possible, in which case a `no method error` is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert{Float64, "foo"}
ERROR: `convert` has no method matching convert{::Type{Float64}, ::ASCIIString}
in convert at base.jl:13
```

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.

Defining New Conversions

To define a new conversion, simply provide a new method for `convert`. That's really all there is to it. For example, the method to convert a number to a boolean is simply this:

```
convert{::Type{Bool}, x::Number} = (x!=0)
```

The type of the first argument of this method is a *singleton type*, `Type{Bool}`, the only instance of which is `Bool`. Thus, this method is only invoked when the first argument is the type value `Bool`. 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 is never used in the function body. In this example, since the type is a singleton, there would never be any reason to use its value within the body. When invoked, the method determines whether a numeric value is true or false as a boolean, by comparing it to zero:

```
julia> convert{Bool, 1}
true

julia> convert{Bool, 0}
false

julia> convert{Bool, 1im}
ERROR: InexactError()
in convert at complex.jl:18

julia> convert{Bool, 0im}
false
```

The method signatures for conversion methods are often quite a bit more involved than this example, especially for parametric types. The example above is meant to be pedagogical, and is not the actual julia behaviour. This is the actual implementation in julia:

```
convert{T<:Real} (::Type{T}, z::Complex) = (imag(z)==0 ? convert(T, real(z)) :
                                           throw(InexactError()))

julia> convert{Bool, lim}
InexactError()
in convert at complex.jl:40
```

Case Study: Rational Conversions

To continue our case study of Julia’s Rational type, here are the conversions declared in `rational.jl`, right after the declaration of the type and its constructors:

```
convert{T<:Integer} (::Type{Rational{T}}, x::Rational) = Rational(convert(T,x.num), convert(T,x.den))
convert{T<:Integer} (::Type{Rational{T}}, x::Integer) = Rational(convert(T,x), convert(T,1))

function convert{T<:Integer} (::Type{Rational{T}}, x::FloatingPoint, tol::Real)
    if isnan(x); return zero(T)//zero(T); end
    if isinf(x); return sign(x)//zero(T); end
    y = x
    a = d = one(T)
    b = c = zero(T)
    while true
        f = convert(T, round(y)); y -= f
        a, b, c, d = f*a+c, f*b+d, a, b
        if y == 0 || abs(a/b-x) <= tol
            return a//b
        end
        y = 1/y
    end
end

convert{T<:Integer} (rt::Type{Rational{T}}, x::FloatingPoint) = convert(rt,x,eps(x))

convert{T<:FloatingPoint} (::Type{T}, x::Rational) = convert(T,x.num)/convert(T,x.den)
convert{T<:Integer} (::Type{T}, x::Rational) = div(convert(T,x.num), convert(T,x.den))
```

The initial four `convert` methods provide conversions to rational types. The first method converts one type of rational to another type of rational by converting the numerator and denominator to the appropriate integer type. The second method does the same conversion for integers by taking the denominator to be 1. The third method implements a standard algorithm for approximating a floating-point number by a ratio of integers to within a given tolerance, and the fourth method applies it, using machine epsilon at the given value as the threshold. In general, one should have `a//b == convert(Rational{Int64}, a/b)`.

The last two `convert` methods provide conversions from rational types to floating-point and integer types. To convert to floating point, one simply converts both numerator and denominator to that floating point type and then divides. To convert to integer, one can use the `div` operator for truncated integer division (rounded towards zero).

1.14.2 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 supertype 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 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 the Julia standard library. 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,Int32}(15,-5)
-3//1
```

```
julia> typeof(ans)
Rational{Int64} (constructor with 1 method)
```

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's standard library:

```
promote_rule{::Type{UInt8}, ::Type{Int8}} = Int
promote_rule{::Type{Char}, ::Type{UInt8}} = Int32
```

As a general rule, Julia promotes integers to *Int* during computation order to avoid overflow. In the latter case, the result type is `Int32` since `Int32` is large enough to contain all possible Unicode code points, and numeric operations on characters always result in plain old integers unless explicitly cast back to characters (see *Characters*). 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, UInt16}
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{T<:Integer}{::Type{Rational{T}}, ::Type{T}} = Rational{T}
promote_rule{T<:Integer, S<:Integer}{::Type{Rational{T}}, ::Type{S}} = Rational{promote_type{T, S}}
promote_rule{T<:Integer, S<:Integer}{::Type{Rational{T}}, ::Type{Rational{S}}} = Rational{promote_type{T, S}}
promote_rule{T<:Integer, S<:FloatingPoint}{::Type{Rational{T}}, ::Type{S}} = promote_type{T, S}
```

The first rule asserts that promotion of a rational number with its own numerator/denominator type, simply promotes to itself. The second 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 third 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 fourth 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 [conversion methods discussed above](#), 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.

1.15 Modules

Modules in Julia are separate global variable workspaces. They are delimited syntactically, inside `module Name` ... `end`. Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

The following example demonstrates the major features of modules. It is not meant to be run, but is shown for illustrative purposes:

```
module MyModule
using Lib

using BigLib: thing1, thing2

import Base.show

importall OtherLib

export MyType, foo

type MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(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` is a syntactic shortcut for `using BigLib.thing1, BigLib.thing2`.

The `import` keyword supports all 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 must be imported using `import` to 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.

The keyword `importall` explicitly imports all names exported by the specified module, as if `import` were individually used on all of them.

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.

1.15.1 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
<code>using MyModule</code>	All export ed names (<code>x</code> and <code>y</code>), <code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>
<code>using MyModule.x,</code> <code>MyModule.p</code>	<code>x</code> and <code>p</code>	
<code>using MyModule:</code> <code>x, p</code>	<code>x</code> and <code>p</code>	
<code>import MyModule</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>
<code>import</code> <code>MyModule.x,</code> <code>MyModule.p</code>	<code>x</code> and <code>p</code>	<code>x</code> and <code>p</code>
<code>import MyModule:</code> <code>x, p</code>	<code>x</code> and <code>p</code>	<code>x</code> and <code>p</code>
<code>importall</code> <code>MyModule</code>	All export ed names (<code>x</code> and <code>y</code>)	<code>x</code> and <code>y</code>

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 `whos()` 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 the standard library (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`, all operators are explicitly imported, since one typically wants to extend operators rather than creating entirely new definitions of them. A module also automatically contains a definition of the `eval` function, which evaluates expressions within the context of that module.

If these definitions are not wanted, modules can be defined using the keyword `baremodule` instead. In terms of `baremodule`, a standard module looks like this:

```
baremodule Mod

using Base

importall Base.Operators

eval(x) = Core.eval(Mod, x)
eval(m,x) = Core.eval(m, x)

...

end
```

Relative and absolute module paths

Given the statement `using Foo`, the system looks for `Foo` within `Main`. 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 module that is not directly available in `Main`. 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 `~/ .juliarc.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`.

Miscellaneous details

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.

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 can be “reserved” for the current module without assigning to it by declaring it as `global x` at the top level. This can be used to prevent name conflicts for globals initialized after load time.

1.16 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.

1.16.1 Program representation

Every Julia program starts life as a string:

```
julia> prog = "1 + 1"
"1 + 1"
```

What happens next?

The next step is to [parse](#) each string into an object called an expression, represented by the Julia type `Expr`:

```
julia> ex1 = parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

`Expr` objects contain three parts:

- a `Symbol` identifying the kind of expression. A symbol is an [interned string](#) identifier (more discussion below).

```
julia> ex1.head
:call
```

- the expression arguments, which may be symbols, other expressions, or literal values:

```
julia> ex1.args
3-element Array{Any,1}:
 :+
 1
 1
```

- finally, the expression result type, which may be annotated by the user or inferred by the compiler (and may be ignored completely for the purposes of this chapter):

```
julia> ex1.typ
Any
```

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
typ: Any
```

Expr objects may also be nested:

```
julia> ex3 = 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
```

Symbols can also be created using `symbol()`, which takes a character or string as its argument:

```
julia> :foo == symbol("foo")
true

julia> symbol("'")
:'
```

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> :(:)
:(:)
```

1.16.2 Expressions and evaluation

Quoting

The second syntactic purpose of the `:` character is to create expression objects without using the explicit `Expr` constructor. This is referred to as *quoting*. The `:` character, followed by paired parentheses around a single statement of Julia code, produces an `Expr` object based on the enclosed code. Here is example of the short form used to quote an arithmetic expression:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)
```



```
julia> typeof(ex)
Expr
```

(to view the structure of this expression, try `ex.head` and `ex.args`, or use `dump()` as above)

Note that equivalent expressions may be constructed using `parse()` or the direct `Expr` form:

```
julia>      :(a + b*c + 1) ==
      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`. Note that this form introduces `QuoteNode` elements to the expression tree, which must be considered when directly manipulating an expression tree generated from `quote` blocks. For other purposes, `:(...)` and `quote .. end` blocks are treated identically.

```
julia> ex = quote
      x = 1
      y = 2
      x + y
    end
quote # none, line 2:
x = 1 # line 3:
y = 2 # line 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 “splicing” or interpolation of literals or expressions into quoted expressions. Interpolation is indicated by the `$` prefix.

In this example, the literal value of `a` is interpolated:

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

In this example, the tuple `(1, 2, 3)` is interpolated as an expression into a conditional test:

```
julia> ex = :(a in $((1,2,3)) )
:($(Expr(:in, :a, :((1,2,3)))))
```

Interpolating symbols into a nested expression requires enclosing each symbol in an enclosing quote block:

```
julia> :( :a in $( :(:a + :b) ) )
      ^^^^^^^^^^
      quoted inner expression
```

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.

`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: a 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: 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

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
```

1.16.3 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
```

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 given at 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
```

This macro takes one argument: name. When @sayhello is encountered, the quoted expression is *expanded* to interpolate the value of the argument into the final expression:

```
julia> @sayhello("human")
Hello, human
```

We can view the quoted return expression using the function `macroexpand()` (**important note:** this is an extremely useful tool for debugging macros):

```
julia> ex = macroexpand( :(@sayhello("human")) )
:(println("Hello, ", "human", "!"))
      ^^^^^^^
      interpolated: now a literal string

julia> typeof(ex)
Expr
```

Hold up: why macros?

We have already seen a function `f (::Expr...) -> Expr` in a previous section. In fact, `macroexpand()` is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code *before* the full program is run. To illustrate the difference, consider the following example:

```
julia> macro twostep(arg)
    println("I execute at parse time. The argument is: ", arg)

    return :(println("I execute at runtime. The argument is: ", $arg))
end

julia> ex = macroexpand( :(@twostep (1, 2, 3)) );
I execute at parse time. The argument is: ((1,2,3))
```

The first call to `println()` is executed when `macroexpand()` is called. The resulting expression contains *only* the second `println`:

```
julia> typeof(ex)
Expr

julia> ex
:(println("I execute at runtime. The argument is: ", $(Expr(:copyast, :((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, ...)
```

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
```

```
julia> @showarg(a)
(:a,)
```

```
julia> @showarg(1+1)
:(1 + 1)
```

```
julia> @showarg(println("Yo!"))
:(println("Yo!"))
```

Building an advanced macro

Here is a simplified definition of Julia's `@assert` macro:

```
macro assert(ex)
    return :($ex ? nothing : error("Assertion failed: ", $(string(ex))))
end
```

This macro can be used like this:

```
julia> @assert 1==1.0

julia> @assert 1==0
ERROR: assertion failed: 1 == 0
in error at error.jl:21
```

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 : error("Assertion failed: ", "1==1.0")
1==0 ? nothing : error("Assertion failed: ", "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 the standard library 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 ellipsis following the last argument:

```
macro assert(ex, msgs...)
    msg_body = isempty(msgs) ? ex : msgs[1]
    msg = string("assertion failed: ", msg_body)
    return :($ex ? nothing : error($msg))
end
```

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()` function:

```
julia> macroexpand(:(@assert a==b))
:(if a == b
    nothing
else
    Base.error("assertion failed: a == b")
end)

julia> macroexpand(:(@assert a==b "a should equal b!"))
:(if a == b
    nothing
else
    Base.error("assertion failed: a should equal b!")
end)
```

There is yet another case that the actual `@assert` macro handles: what if, in addition to printing “a should equal b,” we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., `@assert a==b "a ($a) should equal b ($b)!"`, but this won't work as expected with the above macro. Can you see why? Recall from *string interpolation* that an interpolated string is rewritten to a call to `string()`. Compare:

```
julia> typeof(:("a should equal b"))
ASCIIString (constructor with 2 methods)

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: ASCIIString "a ("
```

```

2: Symbol a
3: ASCIIString ")" should equal b ("
4: Symbol b
5: ASCIIString ")!"
typ: Any

```

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 the standard library, 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 standard library 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:

```
macro zerox()  
    return esc(:(x = 0))  
end  
  
function foo()  
    x = 1  
    @zerox  
    x # is zero  
end
```

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

1.16.4 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, the following code defines a series of operators on three arguments in terms of their 2-argument forms:

```
for op = (:+, :*, :&, :|, :$)  
    eval(quote  
        ($op)(a,b,c) = ($op)($op(a,b),c)  
    end)  
end
```

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 = (:+, :*, :&, :|, :$)  
    eval(:(($op)(a,b,c) = ($op)($op(a,b),c)))  
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 = (:+, :*, :&, :|, :$)  
    @eval ($op)(a,b,c) = ($op)($op(a,b),c)  
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
```

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

```
julia> $a + b
ERROR: unsupported or misplaced expression $
```

1.16.5 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.

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)))
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.

1.17 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.

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 reference. Some technical computing languages pass arrays by value, and this is convenient in many cases. In Julia, modifications made to input arrays within a function will be visible in the parent function. The entire Julia array library ensures that inputs are not modified by library functions. User code, if it needs to exhibit similar behavior, should take care to create a copy of inputs that it may modify.

1.17.1 Arrays

Basic Functions

Function	Description
<code>eltype(A)</code>	the type of the elements contained in <code>A</code>
<code>length(A)</code>	the number of elements in <code>A</code>
<code>ndims(A)</code>	the number of dimensions of <code>A</code>
<code>size(A)</code>	a tuple containing the dimensions of <code>A</code>
<code>size(A, n)</code>	the size of <code>A</code> in a particular dimension
<code>stride(A, k)</code>	the stride (linear index distance between adjacent elements) along dimension <code>k</code>
<code>strides(A)</code>	a tuple of the strides in each dimension

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.

Function	Description
<code>Array{Type, dims...}</code>	an uninitialized dense array
<code>cell{dims...}</code>	an uninitialized cell array (heterogeneous array)
<code>zeros{Type, dims...}</code>	an array of all zeros of specified type, defaults to <code>Float64</code> if <code>type</code> not specified
<code>zeros(A)</code>	an array of all zeros of same element type and shape of <code>A</code>
<code>ones{Type, dims...}</code>	an array of all ones of specified type, defaults to <code>Float64</code> if <code>type</code> not specified
<code>ones(A)</code>	an array of all ones of same element type and shape of <code>A</code>
<code>trues(dims...)</code>	a <code>Bool</code> array with all values <code>true</code>
<code>false{dims...}</code>	a <code>Bool</code> array with all values <code>false</code>
<code>reshape(A, dims...)</code>	an array with the same data as the given array, but with different dimensions.
<code>copy(A)</code>	copy <code>A</code>
<code>deepcopy(A)</code>	copy <code>A</code> , recursively copying its elements
<code>similar(A, element_type, dims...)</code>	an uninitialized array of the same type as the given array (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 <code>A</code> if omitted.
<code>reinterpret{Type, A}</code>	an array with the same binary data as the given array, but with the specified element type
<code>rand(dims)</code>	Array of <code>Float64</code> s with random, iid[#]_ and uniformly distributed values in the half-open interval <code>[0, 1)</code>
<code>randn(dims)</code>	Array of <code>Float64</code> s with random, iid and standard normally distributed random values
<code>eye(n)</code>	<code>n</code> -by- <code>n</code> identity matrix
<code>eye(m, n)</code>	<code>m</code> -by- <code>n</code> identity matrix
<code>linspace(start, stop, n)</code>	vector of <code>n</code> linearly-spaced elements from <code>start</code> to <code>stop</code>
<code>fill!(A, x)</code>	fill the array <code>A</code> with the value <code>x</code>
<code>fill(x, dims)</code>	create an array filled with the value <code>x</code>

Concatenation

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

Function	Description
<code>cat(k, A...)</code>	concatenate input n-d arrays along the dimension <code>k</code>
<code>vcat(A...)</code>	shorthand for <code>cat(1, A...)</code>
<code>hcat(A...)</code>	shorthand for <code>cat(2, A...)</code>

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

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

Expression	Calls
<code>[A B C ...]</code>	<code>hcat()</code>
<code>[A, B, C, ...]</code>	<code>vcat()</code>
<code>[A B; C D; ...]</code>	<code>hvcat()</code>

`hvcat()` concatenates in both dimension 1 (with semicolons) and dimension 2 (with spaces).

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.

Special syntax is available for constructing arrays with element type `Any`:

Expression	Yields
<code>{A B C ...}</code>	A 1xN <code>Any</code> array
<code>{A, B, C, ...}</code>	A 1-d <code>Any</code> array (vector)
<code>{A B; C D; ...}</code>	A 2-d <code>Any</code> array

Note that this form does not do any concatenation; each argument becomes an element of the resulting array.

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> const 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
```

Note: In the above example, `x` is declared as constant because type inference in Julia does not work as well on non-constant global variables.

The resulting array type is inferred from the expression; in order to control the type explicitly, the type can be prepended to the comprehension. For example, in the above example we could have avoided declaring `x` as constant, and ensured that the result is of type `Float64` by writing:

```
Float64[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

Using curly brackets instead of square brackets is a shorthand notation for an array of type `Any`:

```
julia> { i/2 for i = 1:3 }
3-element Array{Any,1}:
 0.5
 1.0
 1.5
```

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:

1. A scalar value
2. A Range of the form `:`, `a:b`, or `a:b:c`
3. An arbitrary integer vector, including the empty vector `[]`
4. A boolean vector

The result `X` generally has dimensions $(\text{length}(I_1), \text{length}(I_2), \dots, \text{length}(I_n))$, with location (i_1, i_2, \dots, i_n) of `X` containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`. Trailing dimensions indexed with scalars are dropped. For example, the dimensions of `A[I, 1]` will be $(\text{length}(I),)$. Boolean vectors are first transformed with `find`; the size of a dimension indexed by a boolean vector will be the number of true values in the vector.

Indexing syntax is equivalent to a call to `getindex`:

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

Example:

```
julia> x = reshape(1:16, 4, 4)
4x4 Array{Int64,2}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16
```

```
julia> x[2:3, 2:end-1]
2x2 Array{Int64,2}:
 6 10
 7 11
```

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
```

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:

1. A scalar value
2. A Range of the form `:`, `a:b`, or `a:b:c`
3. An arbitrary integer vector, including the empty vector `[]`
4. A boolean vector

If `X` is an array, its size must be `(length(I_1), length(I_2), ..., length(I_n))`, and the value in location `i_1, i_2, ..., i_n` of `A` is overwritten with the value `X[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`. If `X` is not an array, its value is written to all referenced locations of `A`.

A boolean vector used as an index behaves as in `getindex()` (it is first transformed with `find()`).

Index assignment syntax is equivalent to a call to `setindex!()`:

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

Example:

```
julia> x = reshape(1:9, 3, 3)
3x3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[1:2, 2:3] = -1
-1

julia> x
3x3 Array{Int64,2}:
-1 -1 -1
-1 -1 -1
 6  9
```

```
1  -1  -1
2  -1  -1
3   6   9
```

Vectorized Operators and Functions

The following operators are supported for arrays. The dot version of a binary operator should be used for elementwise operations.

1. Unary arithmetic — `-`, `+`, `!`
2. Binary arithmetic — `+`, `-`, `*`, `.*`, `/`, `./`, `\`, `.\`, `^`, `.^`, `div`, `mod`
3. Comparison — `==`, `!=`, `<`, `<=`, `>`, `>=`
4. Unary Boolean or bitwise — `~`
5. Binary Boolean or bitwise — `&`, `|`, `$`

Some operators without dots operate elementwise anyway when one argument is a scalar. These operators are `*`, `+`, `-`, and the bitwise operators. The operators `/` and `\` operate elementwise when the denominator is a scalar.

Note that comparisons such as `==` operate on whole arrays, giving a single boolean answer. Use dot operators for elementwise comparisons.

The following built-in functions are also vectorized, whereby the functions act elementwise:

```
abs abs2 angle cbrt
airy airyai airyaiprime airybi airybiprime airyprime
acos acosh asin asinh atan atan2 atanh
acsc acsch asec asech acot acoth
cos cospi cosh sin sinpi sinh tan tanh sinc cosc
csc csch sec sech cot coth
acosd asind atand asecd acscd acotd
cosd sind tand secd cscd cotd
besselh besseli besselj besselj0 besselj1 besselk bessely bessely0 bessely1
exp erf erfc erfinv erfcinv exp2 expm1
beta dawson digamma erfcx erfi
exponent eta zeta gamma
hankelh1 hankelh2
ceil floor round trunc
iceil ifloor iround itrunc
isfinite isinf isnan
lbeta lfact lgamma
log log10 loglp log2
copysign max min significand
sqrt hypot
```

Note that there is a difference between `min()` and `max()`, which operate elementwise over multiple array arguments, and `minimum()` and `maximum()`, which find the smallest and largest values within an array.

Julia provides the `@vectorize_larg()` and `@vectorize_2arg()` macros to automatically vectorize any function of one or two arguments respectively. Each of these takes two arguments, namely the `Type` of argument (which is usually chosen to be the most general possible) and the name of the function to vectorize. Here is a simple example:

```
julia> square(x) = x^2
square (generic function with 1 method)

julia> @vectorize_larg Number square
square (generic function with 4 methods)
```

```
julia> methods(square)
# 4 methods for generic function "square":
square{T<:Number} (::AbstractArray{T<:Number,1}) at operators.jl:359
square{T<:Number} (::AbstractArray{T<:Number,2}) at operators.jl:360
square{T<:Number} (::AbstractArray{T<:Number,N}) at operators.jl:362
square(x) at none:1

julia> square([1 2 4; 5 6 7])
2x3 Array{Int64,2}:
 1  4 16
25 36 49
```

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> a = rand(2,1); A = rand(2,3);

julia> repmat(a,1,3)+A
2x3 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> broadcast(+, a, A)
2x3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1x2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2x2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

Elementwise operators such as `.+` and `.*` perform broadcasting if necessary. There is also a `broadcast!()` function to specify an explicit destination, and `broadcast_getindex()` and `broadcast_setindex!()` that broadcast the indices before indexing.

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 $\tilde{O}(1)$ 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()`.

`DenseArray` is an abstract subtype of `AbstractArray` intended to include all arrays that are laid out at regular offsets in memory, and which can therefore be passed to external C and Fortran functions expecting this memory layout. Subtypes should provide a method `stride(A,k)` that returns the “stride” of dimension `k`: 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 `convert{Ptr{T}, A}` is provided, the memory layout should correspond in the same way to these strides.

The `Array{T,N}` type is a specific instance of `DenseArray` where elements are stored in column-major order (see additional notes in *Performance Tips*). `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Specific operations such as scalar indexing, assignment, and a few other basic storage-specific operations are all that have to be implemented for `Array`, so that the rest of the array library can be implemented in a generic manner.

`SubArray` is a specialization of `AbstractArray` that performs indexing by reference rather than by copying. A `SubArray` is created with the `sub()` function, which is called the same way as `getindex()` (with an array and a series of index arguments). The result of `sub()` looks the same as the result of `getindex()`, except the data is left in place. `sub()` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly.

`StridedVector` and `StridedMatrix` are convenient aliases defined to make it possible for Julia to call a wider range of BLAS and LAPACK functions by passing them either `Array` or `SubArray` objects, and thus saving inefficiencies from memory allocation and copying.

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> a = rand(10,10)
10x10 Array{Float64,2}:
 0.561255  0.226678  0.203391  0.308912  ...  0.750307  0.235023  0.217964
 0.718915  0.537192  0.556946  0.996234  ...  0.666232  0.509423  0.660788
 0.493501  0.0565622 0.118392  0.493498  ...  0.262048  0.940693  0.252965
 0.0470779 0.736979  0.264822  0.228787  ...  0.161441  0.897023  0.567641
 0.343935  0.32327  0.795673  0.452242  ...  0.468819  0.628507  0.511528
 0.935597  0.991511 0.571297  0.74485  ...  0.84589  0.178834  0.284413
 0.160706  0.672252 0.133158  0.65554  ...  0.371826  0.770628  0.0531208
 0.306617  0.836126 0.301198  0.0224702 ...  0.39344  0.0370205 0.536062
 0.890947  0.168877 0.32002  0.486136  ...  0.096078  0.172048  0.77672
 0.507762  0.573567 0.220124  0.165816  ...  0.211049  0.433277  0.539476

julia> b = sub(a, 2:2:8,2:2:4)
4x2 SubArray{Float64,2,Array{Float64,2},(StepRange{Int64,Int64},StepRange{Int64,Int64})}:
 0.537192  0.996234
 0.736979  0.228787
 0.991511  0.74485
 0.836126  0.0224702

julia> (q,r) = qr(b);

julia> q
4x2 Array{Float64,2}:
```

```
-0.338809  0.78934
-0.464815 -0.230274
-0.625349  0.194538
-0.527347 -0.534856

julia> r
2x2 Array{Float64,2}:
-1.58553 -0.921517
 0.0      0.866567
```

1.17.2 Sparse Matrices

Sparse matrices are matrices that contain enough zeros that storing them in a special data structure leads to savings in space and execution time. Sparse matrices may be used when operations on the sparse representation of a matrix lead to considerable gains in either time or space when compared to performing the same operations on a dense matrix.

Compressed Sparse Column (CSC) 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 nonzero values, and `Ti` is the integer type for storing column pointers and row indices.:

```
type SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int          # Number of rows
    n::Int          # Number of columns
    colptr::Vector{Ti} # Column i is in colptr[i):(colptr[i+1]-1)
    rowval::Vector{Ti} # Row values of nonzeros
    nzval::Vector{Tv}  # Nonzero values
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of nonzero values 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 actual values that are nonzero, use `countnz()`, which inspects every stored element of a sparse matrix.

Sparse matrix constructors

The simplest way to create sparse matrices is to use functions equivalent to the `zeros()` and `eye()` functions that Julia provides for working with dense matrices. To produce sparse matrices instead, you can use the same names with an `sp` prefix:

```
julia> spzeros(3,5)
3x5 sparse matrix with 0 Float64 entries:
```

```
julia> speye(3,5)
3x5 sparse matrix with 3 Float64 entries:
 [1, 1] = 1.0
 [2, 2] = 1.0
 [3, 3] = 1.0
```

The `sparse()` function is often a handy way to construct sparse matrices. It takes as its input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of nonzero values. `sparse(I, J, V)` constructs a sparse matrix such that `S[I[k], J[k]] = V[k]`.

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];
```

```
julia> S = sparse(I,J,V)
5x18 sparse matrix with 4 Int64 entries:
 [1, 4] = 1
 [4, 7] = 2
 [5, 9] = 3
 [3, 18] = -5
```

The inverse of the `sparse()` function is `findn()`, which retrieves the inputs used to create the sparse matrix.

```
julia> findn(S)
([1,4,5,3], [4,7,9,18])

julia> findnz(S)
([1,4,5,3], [4,7,9,18], [1,2,3,-5])
```

Another way to create sparse matrices is to convert a dense matrix into a sparse matrix using the `sparse()` function:

```
julia> sparse(eye(5))
5x5 sparse matrix with 5 Float64 entries:
 [1, 1] = 1.0
 [2, 2] = 1.0
 [3, 3] = 1.0
 [4, 4] = 1.0
 [5, 5] = 1.0
```

You can go in the other direction using the `full()` function. The `issparse()` function can be used to query if a matrix is sparse.

```
julia> issparse(speye(5))
true
```

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 non-zeroes or the structure in the dense vectors `(I, J, V)`, and then reconstruct the sparse matrix.

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 Matrices* section of the standard library reference.

Sparse	Dense	Description
<code>spzeros(m, n)</code>	<code>zeros(m, n)</code>	Creates a m -by- n matrix of zeros. (<code>spzeros(m, n)</code> is empty.)
<code>spones(S)</code>	<code>ones(m, n)</code>	Creates a matrix filled with ones. Unlike the dense version, <code>spones()</code> has the same sparsity pattern as S .
<code>speye(n)</code>	<code>eye(n)</code>	Creates a n -by- n identity matrix.
<code>full(S)</code>	<code>sparse(A)</code>	Interconverts between dense and sparse formats.
<code>sprand(m, n, d)</code>	<code>rand(m, n)</code>	Creates a m -by- n random matrix (of density d) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
<code>sprandn(m, n, d)</code>	<code>randn(m, n)</code>	Creates a m -by- n random matrix (of density d) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
<code>sprandn(m, n, d, X)</code>	<code>randn(m, n, X)</code>	Creates a m -by- n random matrix (of density d) with iid non-zero elements distributed according to the X distribution. (Requires the <code>Distributions</code> package.)
<code>sprandbool(m, n, d)</code>	<code>randbool(m, n)</code>	Creates a m -by- n random matrix (of density d) with non-zero <code>Bool</code> elements with probability d ($d = 0.5$ for <code>randbool()</code>).

1.18 Linear algebra

1.18.1 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 *Linear Algebra* section of the standard library documentation.

Cholesky	Cholesky factorization
CholeskyPivoted	Pivoted Cholesky factorization
LU	LU factorization
LUTridiagonal	LU factorization for Tridiagonal matrices
UmfpackLU	LU factorization for sparse matrices (computed by UMFPack)
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

1.18.2 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.

Hermitian	Hermitian matrix
Triangular	Upper/lower triangular matrix
Tridiagonal	Tridiagonal matrix
SymTridiagonal	Symmetric tridiagonal matrix
Bidiagonal	Upper/lower bidiagonal matrix
Diagonal	Diagonal matrix
UniformScaling	Uniform scaling operator

Elementary operations

Matrix type	+	-	*	\	Other functions with optimized methods
Hermitian				MV	<code>inv()</code> , <code>sqrtm()</code> , <code>expm()</code>
Triangular			MV	MV	<code>inv()</code> , <code>det()</code>
SymTridiagonal	M	M	MS	MV	<code>eigmax()</code> , <code>eigmin()</code>
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	<code>inv()</code> , <code>det()</code> , <code>logdet()</code> , <code>/()</code>
UniformScaling	M	M	MVS	MVS	<code>/()</code>

Legend:

M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Matrix factorizations

Matrix type	LAPACK	<code>eig()</code>	<code>eigvals()</code>	<code>eigvecs()</code>	<code>svd()</code>	<code>svdvals()</code>
Hermitian	HE		ARI			
Triangular	TR					
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

Legend:

A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. <code>eigvals(M)</code>
R (range)	An optimized method to find the i_1^{th} through the i_h^{th} characteristic values are available	<code>eigvals(M, i1, ih)</code>
I (interval)	An optimized method to find the characteristic values in the interval $[v_l, v_h]$ is available	<code>eigvals(M, vl, vh)</code>
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values $x = [x_1, x_2, \dots]$ is available	<code>eigvecs(M, x)</code>

The uniform scaling operator

A `UniformScaling` operator represents a scalar times the identity operator, $\lambda \cdot \mathbf{I}$. The identity operator \mathbf{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 :class: `I` is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

1.19 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.

1.19.1 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")
Hello World
```

```
julia> read(STDIN, Char)
```

```
'\n'
```

Note that I pressed enter 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> readbytes(STDIN, 4)
abcd
4-element Array{UInt8, 1}:
 0x61
 0x62
 0x63
 0x64
```

or if we had wanted to read the entire line instead:

```
julia> readline(STDIN)
abcd
"abcd\n"
```

Note that depending on your terminal settings, your TTY may be line buffered and might thus require an additional enter before the data is sent to Julia.

To read every line from `STDIN` you can use `eachline()`:

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

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

```
while !eof(STDIN)
    x = read(STDIN, Char)
    println("Found: $x")
end
```

1.19.2 Text I/O

Note that the write method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

```
julia> write(STDOUT, 0x61)
a
```

For text I/O, use the `print()` or `show()` methods, depending on your needs (see the standard library reference for a detailed discussion of the difference between the two):

```
julia> print(STDOUT, 0x61)
97
```

1.19.3 Working with Files

Like many other environments, Julia has an `open()` function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Array{Union{ASCIIString, UTF8String}, 1}:
"Hello, World!\n"
```

If you want to write to a file, you can open it with the write ("`w`") flag:

```
julia> f = open("hello.txt", "w")
IOStream(<file hello.txt>)

julia> write(f, "Hello again.")
12
```

If you examine the contents of `hello.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IOStream` must be closed before the write is actually flushed to disk:

```
julia> close(f)
```

Examining `hello.txt` again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of `open()` which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again. For example, given a function:

```
function read_and_capitalize(f::IOStream)
    return uppercase(readall(f))
end
```

You can call:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

to open `hello.txt`, call `read_and_capitalize` on it, close `hello.txt` and return the capitalized contents.

To avoid even having to define a named function, you can use the `do` syntax, which creates an anonymous function on the fly:

```
julia> open("hello.txt") do f
    uppercase(readall(f))
end
"HELLO AGAIN."
```

1.19.4 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. Let's first create a simple server:

```
julia> @async begin
    server = listen(2000)
    while true
        sock = accept(server)
        println("Hello World\n")
    end
end
```

Task

```
julia>
```

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)
TcpServer(active)

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

julia> listen(ip "::1",2000) # Listens on localhost:2000 (IPv6)
TcpServer(active)
```



```
julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
TcpServer(active)

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

julia> listen("testsocket") # Listens on a domain socket/named pipe
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 domain socket (UNIX). The difference 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 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))
        end
    end
end

Task

julia> clientside=connect(2001)
TcpSocket(open, 0 bytes waiting)

julia> @async while true
    write(STDOUT, readline(clientside))
end

julia> println(clientside, "Hello World from the Echo Server")

julia> Hello World from the Echo Server
```

As with other streams, use `close()` to disconnect the socket:

```
julia> close(clientside)
```

1.19.5 Resolving IP Addresses

One of the `connect()` methods that does not follow the `listen()` methods is `connect(host::ASCIIString, 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(open, 0 bytes waiting)
```

At the base of this functionality is `getaddrinfo()`, which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
IPv4(74.125.226.225)
```

1.20 Parallel Computing

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 MPI³. 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.

Parallel 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. A remote call returns a remote reference 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 its remote reference, and you can obtain the full value of the result using `fetch()`. You can store a value to a remote reference using `put!()`.

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 cores on the machine.

```
$ ./julia -p 2
```

```
julia> r = remotecall(2, rand, 2, 2)
RemoteRef{2,1,5}
```

```
julia> fetch(r)
2x2 Float64 Array:
 0.60401  0.501111
```

³ 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://www.mpi-forum.org/docs>.

```
0.174572 0.157411

julia> s = @spawnat 2 1 .+ fetch(r)
RemoteRef(2,1,7)

julia> fetch(s)
2x2 Float64 Array:
 1.60401  1.50111
 1.17457  1.15741
```

The first argument to `remotecall()` is the index of the process that will do the work. 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 function to call, and the remaining arguments will be passed to this function. 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 remote references, `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(2, getindex, r, 1, 1)
0.10824216411304866
```

Remember that `getindex(r, 1, 1)` is *equivalent* to `r[1, 1]`, so this call fetches the first element of the remote reference `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)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

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.

(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.)

1.20.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

```
julia> function rand2(dims...)
    return 2*rand(dims...)
end

julia> rand2(2,2)
2x2 Float64 Array:
```

```
0.153756  0.368514
1.15119   0.918912

julia> @spawn rand2(2,2)
RemoteRef(1,1,1)

julia> @spawn rand2(2,2)
RemoteRef(2,1,2)

julia> exception on 2: in anonymous: rand2 not defined
```

Process 1 knew about the function `rand2`, but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, `"DummyModule.jl"`, containing the following code:

```
module DummyModule

export MyType, f

type MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

Starting julia with `julia -p 2`, you can use this to verify the following:

- `include("DummyModule.jl")` loads the file on just a single process (whichever one executes the statement).
- `using DummyModule` causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement.
- As long as `DummyModule` is loaded on process 2, commands like

```
rr = RemoteRef(2)
put!(rr, MyType(7))
```

allow you to store an object of type `MyType` on process 2 even if `DummyModule` is not in scope on process 2.

You can force a command to run on all processes using the `@everywhere` macro. Consequently, an easy way to load *and* use a package on all processes is:

```
@everywhere using DummyModule
```

`@everywhere` can also be used to directly define a function on all processes:

```
julia> @everywhere id = myid()

julia> remotecall_fetch(2, ()->id)
2
```

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

Each process has an associated identifier. The process providing the interactive Julia prompt always has an id equal to 1, as would the Julia process running the driver script in the example above. 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.

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 `--machinefile` 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.

Other types of clusters can be supported by writing your own custom `ClusterManager`, as described below in the *ClusterManagers* section.

1.20.2 Data Movement

Sending messages and moving data constitute most of the overhead in a parallel 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 parallel 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
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
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)` then 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.

1.20.3 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 += randbool()
    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:

```
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

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*, which can be written in Julia like this:

```
nheads = @parallel (+) for i=1:200000000
    int(randbool())
end
```

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case `(+)`). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```
a = zeros(100000)
@parallel for i=1:100000
    a[i] = i
end
```

Notice that the reduction operator can be omitted if it is not needed. However, 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, distributed arrays can be used to get around this limitation, as we will see in the next section.

Using “outside” variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)
@parallel (+) 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.

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:

```
M = {rand(1000,1000) for i=1:10}
pmap(svd, M)
```

Julia’s `pmap()` is designed for the case where each function call does a large amount of work. In contrast, `@parallel for` can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both `pmap()` and `@parallel for` for the parallel computation. In case of `@parallel for`, the final reduction is done on the calling process.

1.20.4 Synchronization With Remote References

1.20.5 Scheduling

Julia’s parallel programming platform uses *Tasks (aka Coroutines)* to switch among multiple computations. 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.

As an example, consider computing the singular values of matrices of different sizes:

```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}
pmap(svd, M)
```

If one process handles both 800x800 matrices and another handles both 600x600 matrices, we will not get as much scalability as we could. The solution is to make a local task to “feed” work to each process when it completes its current task. This can be seen in the implementation of `pmap()`:

```
function pmap(f, lst)
    np = nprocs() # determine the number of processes available
    n = length(lst)
    results = cell{n}
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(p, f, lst[idx])
                    end
                end
            end
        end
    end
end
```

```
        end
    end
    results
end
```

`@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 indexes. 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. The feeder tasks are able to share state via `nextidx()` because they all run on the same process. No locking is required, since the threads are scheduled cooperatively and not preemptively. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch()` is called.

1.20.6 Distributed Arrays

Large computations are often organized around large arrays of data. In these cases, a particularly natural way to obtain parallelism is to distribute arrays among several processes. This combines the memory resources of multiple machines, allowing use of arrays too large to fit on one machine. Each process operates on the part of the array it owns, providing a ready answer to the question of how a program should be divided among machines.

Julia distributed arrays are implemented by the `DArray` type. A `DArray` has an element type and dimensions just like an `Array`. A `DArray` can also use arbitrary array-like types to represent the local chunks that store actual data. The data in a `DArray` is distributed by dividing the index space into some number of blocks in each dimension.

Common kinds of arrays can be constructed with functions beginning with `d`:

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x,100,100,10)
```

In the last case, each element will be initialized to the specified value `x`. These functions automatically pick a distribution for you. For more control, you can specify which processes to use, and how the data should be distributed:

```
dzeros((100,100), workers()[1:4], [1,4])
```

The second argument specifies that the array should be created on the first four workers. When dividing data among a large number of processes, one often sees diminishing returns in performance. Placing `DArrays` on a subset of processes allows multiple `DArray` computations to happen at once, with a higher ratio of work to communication on each process.

The third argument specifies a distribution; the `n`th element of this array specifies how many pieces dimension `n` should be divided into. In this example the first dimension will not be divided, and the second dimension will be divided into 4 pieces. Therefore each local chunk will be of size `(100, 25)`. Note that the product of the distribution array must equal the number of processes.

`distribute(a::Array)` converts a local array to a distributed array.

`localpart(a::DArray)` obtains the locally-stored portion of a `DArray`.

`localindexes(a::DArray)` gives a tuple of the index ranges owned by the local process.

`convert(Array, a::DArray)` brings all the data to the local process.

Indexing a `DArray` (square brackets) with ranges of indexes always creates a `SubArray`, not copying any data.

1.20.7 Constructing Distributed Arrays

The primitive `DArray` constructor has the following somewhat elaborate signature:

```
DArray{init, dims[, procs, dist]}
```

`init` is a function that accepts a tuple of index ranges. This function should allocate a local chunk of the distributed array and initialize it for the specified indices. `dims` is the overall size of the distributed array. `procs` optionally specifies a vector of process IDs to use. `dist` is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.

The last two arguments are optional, and defaults will be used if they are omitted.

As an example, here is how to turn the local array constructor `fill()` into a distributed array constructor:

```
dfill(v, args...) = DArray{I->fill(v, map(length,I)), args...}
```

In this case the `init` function only needs to call `fill()` with the dimensions of the local piece it is creating.

1.20.8 Distributed Array Operations

At this time, distributed arrays do not have much functionality. Their major utility is allowing communication to be done via array indexing, which is convenient for many problems. As an example, consider implementing the “life” cellular automaton, where each cell in a grid is updated according to its neighboring cells. To compute a chunk of the result of one iteration, each process needs the immediate neighbor cells of its local chunk. The following code accomplishes this:

```
function life_step(d::DArray)
    DArray{size(d),procs(d)} do I
        top    = mod(first(I[1])-2,size(d,1))+1
        bot    = mod( last(I[1])  ,size(d,1))+1
        left   = mod(first(I[2])-2,size(d,2))+1
        right  = mod( last(I[2])  ,size(d,2))+1

        old = Array{Bool, length(I[1])+2, length(I[2])+2}
        old[1      , 1      ] = d[top , left]   # left side
        old[2:end-1, 1      ] = d[I[1], left]
        old[end     , 1      ] = d[bot , left]
        old[1      , 2:end-1] = d[top , I[2]]
        old[2:end-1, 2:end-1] = d[I[1], I[2]]   # middle
        old[end     , 2:end-1] = d[bot , I[2]]
        old[1      , end     ] = d[top , right] # right side
        old[2:end-1, end     ] = d[I[1], right]
        old[end     , end     ] = d[bot , right]

        life_rule(old)
    end
end
```

As you can see, we use a series of indexing expressions to fetch data into a local array `old`. Note that the `do` block syntax is convenient for passing `init` functions to the `DArray` constructor. Next, the serial function `life_rule` is called to apply the update rules to the data, yielding the needed `DArray` chunk. Nothing about `life_rule` is `DArray`-specific, but we list it here for completeness:

```
function life_rule(old)
    m, n = size(old)
    new = similar(old, m-2, n-2)
    for j = 2:n-1
```

```
    for i = 2:m-1
        nc = +(old[i-1,j-1], old[i-1,j], old[i-1,j+1],
               old[i ,j-1],          old[i ,j+1],
               old[i+1,j-1], old[i+1,j], old[i+1,j+1])
        new[i-1,j-1] = (nc == 3 || nc == 2 && old[i,j])
    end
end
new
end
```

1.20.9 Shared Arrays (Experimental)

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.

`SharedArray` indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on `SharedArrays`, albeit in single-process mode. In cases where an algorithm insists on an `Array` input, the underlying array can be retrieved from a `SharedArray` by calling `sdata()`. For other `AbstractArray` types, `sdata` just returns the object itself, so it’s safe to use `sdata()` on any `Array`-type object.

The constructor for a shared array is of the form:

```
SharedArray{T::Type, dims::NTuple; init=false, pids=Int[])
```

which creates a shared array of a bitstype `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 arrange it so that each worker runs the `init` function on a distinct portion of the array, thereby parallelizing initialization.

Here’s a brief example:

```
julia> addprocs(3)
3-element Array{Any,1}:
 2
 3
 4

julia> S = SharedArray{Int, (3,4), init = S -> S[localindexes(S)] = myid()}
3x4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3x4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

`localindexes()` provides disjoint one-dimensional ranges of indexes, 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, (3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] = myid()}
3x4 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(p, fill!, 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.

1.20.10 ClusterManagers

Julia worker processes can also be spawned on arbitrary machines, enabling Julia's natural parallelism to function quite transparently in a cluster environment. The `ClusterManager` interface provides a way to specify a means to launch and manage worker processes. For example, `ssh` clusters are also implemented using a `ClusterManager`:

```
immutable SSHManager <: ClusterManager
    launch::Function
    manage::Function
    machines::AbstractVector

    SSHManager(; machines=[]) = new(launch_ssh_workers, manage_ssh_workers, machines)
end

function launch_ssh_workers(cman::SSHManager, np::Integer, config::Dict)
    ...
end

function manage_ssh_workers(id::Integer, config::Dict, op::Symbol)
    ...
end
```

where `launch_ssh_workers()` is responsible for instantiating new Julia processes and `manage_ssh_workers()` provides a means to manage those processes, e.g. for sending interrupt signals. New processes can then be added at runtime using `addprocs()`:

```
addprocs(5, cman=LocalManager())
```

which specifies a number of processes to add and a `ClusterManager` to use for launching those processes.

1.21 Interacting With Julia

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: http://docs.julialang.org
      | Type "help()" to list help topics
      |
      | Version 0.3.0-prerelease+2834 (2014-04-30 03:13 UTC)
      | Commit 64f437b (0 days old master)
      | x86_64-apple-darwin13.1.0

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.

1.21.1 The different prompt modes

The Julian mode

The REPL has four main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`. It is here that you can enter Julia expressions. Hitting return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

```
julia> string(1 + 2)
"3"
```

There are a number useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable `ans`. A trailing semicolon on the line can be used as a flag to suppress showing the result.

```
julia> string(3 * 4);

julia> ans
"12"
```

Help mode

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing `?`. Julia will attempt to print help or documentation for anything entered in help mode:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help>

help> string
Base.string(xs...)
```

Create a string from any values **using** the `"print"` function.

In addition to function names, complete function calls may be entered to see which method is called for the given argument(s). Macros, types and variables can also be queried:

```
help> string(1)
string(x::Union{Int16, Int128, Int8, Int32, Int64}) at string.jl:1553

help> @printf
```

```
Base.@printf([io::IOStream], "%Fmt", args...)
```

Print `arg(s)` **using** C `"printf()"` style format specification string. Optionally, an `IOStream` may be passed as the first argument to redirect output.

```
help> String
```

```
DataType    : String
```

```
  supertype: Any
```

```
  subtypes : {DirectIndexString,GenericString,RepString,RevString{T<:String},RopeString,SubString{T<
```

Help mode can be exited by pressing backspace at the beginning of the line.

Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as `?` entered help mode when at the beginning of the line, a semicolon (`;`) will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>
```

```
shell> echo hello
```

```
hello
```

Search modes

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an incremental search through the previous history, type `^R` — the control key together with the `r` key. The prompt will change to (reverse-i-search) `'' :`, and as you type the search query will appear in the quotes. The most recent result that matches the query will dynamically update to the right of the colon as more is typed. To find an older result using the same query, simply type `^R` again.

Just as `^R` is a reverse search, `^S` is a forward search, with the prompt (i-search) `'' :`. The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

1.21.2 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).

Program control	
<code>^D</code>	Exit (when buffer is empty)
<code>^C</code>	Interrupt or cancel
Return/Enter, <code>^J</code>	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
<code>? or ;</code>	Enter help or shell mode (when at start of a line)
<code>^R, ^S</code>	Incremental history search, described above
Cursor movement	
Right arrow, <code>^F</code>	Move right one character
Left arrow, <code>^B</code>	Move left one character
Home, <code>^A</code>	Move to beginning of line
End, <code>^E</code>	Move to end of line
<code>^P</code>	Change to the previous or next history entry
<code>^N</code>	Change to the next history entry
Up arrow	Move up one line (or to the previous history entry)
Down arrow	Move down one line (or to the next history entry)
Page-up	Change to the previous history entry that matches the text before the cursor
Page-down	Change to the next history entry that matches the text before the cursor
meta-F	Move right one word
meta-B	Move left one word
Editing	
Backspace, <code>^H</code>	Delete the previous character
Delete, <code>^D</code>	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
meta-D	Forward delete the next word
<code>^W</code>	Delete previous text up to the nearest whitespace
<code>^K</code>	“Kill” to end of line, placing the text in a buffer
<code>^Y</code>	“Yank” insert the text from the kill buffer
<code>^T</code>	Transpose the characters about the cursor
Delete, <code>^D</code>	Forward delete one character (when buffer has text)

1.21.3 Tab completion

In both the Julian and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

```
julia> stri
stride      strides      string      stringmime  strip

julia> Stri
StridedArray  StridedVecOrMat  String
StridedMatrix  StridedVector
```

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e1 = [1,0]
2-element Array{Int64,1}:
 1
 0
```

```

julia> e\1[TAB] = [1 0]
julia> e¹ = [1 0]
1x2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2      # √ is equivalent to the sqrt() function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)

julia> \h[TAB]
\hat          \heartsuit      \hksearrow      \hookleftarrow  \hslash
\hbar         \hermitconjmatrix \hkswarrow      \hookrightarrow  \hspace

```

A full list of tab-completions can be found in the *Unicode Input* section of the manual.

1.22 Running External Programs

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

```

julia> `echo hello`
`echo hello`

```

differs in several aspects from the behavior in various shells, Perl, or Ruby:

- Instead of immediately running the command, backticks create a `Cmd` object to represent the command. You can use this object to connect the command to others via pipes, run it, and read or write to it.
- When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to `STDOUT` as it would using `libc`'s `system` call.
- The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as `julia`'s immediate child process, using `fork` and `exec` calls.

Here's a simple example of running an external program:

```

julia> run(`echo hello`)
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, `readall()` can be used instead:

```

julia> a=readall(`echo hello`)
"hello\n"

julia> (chomp(a)) == "hello"
true

```

More generally, you can use `open()` to read from or write to an external command. For example:

```

julia> open(`less`, "w", STDOUT) do io
    for i = 1:1000

```

```
        println(io, i)
    end
end
```

1.22.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use `$` for interpolation much as you would in a string literal (see *Strings*):

```
julia> file = "/etc/passwd"
"/etc/passwd"
```

```
julia> `sort $file`
`sort /etc/passwd`
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted to sort the contents of the file `/Volumes/External HD/data.csv`. Let's try it:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"
```

```
julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

How did the file name get quoted? Julia knows that `file` is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of `file` is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"
```

```
julia> name = "data"
"data"
```

```
julia> ext = "csv"
"csv"
```

```
julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

As you can see, the space in the `path` variable is appropriately escaped. But what if you *want* to interpolate multiple words? In that case, just use an array (or any other iterable container):

```
julia> files = ["/etc/passwd", "/Volumes/External HD/data.csv"]
2-element ASCIIString Array:
"/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 ASCIIString Array:
"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 ASCIIString Array:
"foo"
"bar"
"baz"

julia> exts = ["aux", "log"]
2-element ASCIIString Array:
"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`
```

1.22.2 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> `perl -le '$|=1; for (0..3) { print }`  
`perl -le '$|=1; for (0..3) { print }`  
  
julia> run(ans)  
0  
1  
2  
3  
  
julia> first = "A"; second = "B";  
  
julia> `perl -le 'print for @ARGV' "1: $first" "2: $second"`  
`perl -le 'print for @ARGV' '1: A' '2: B'`  
  
julia> run(ans)  
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.

1.22.3 Pipelines

Shell metacharacters, such as `|`, `&`, and `>`, are not special inside of Julia’s backticks: unlike in the shell, inside of Julia’s backticks, a pipe is always just a pipe:

```
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”. Inside of backticks, a “|” is just a literal pipe character. How, then, does one construct a pipeline? Instead of using “|” inside of backticks, one uses Julia’s `|>` operator between `Cmd` objects:

```
julia> run(`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(`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. Note that `|>` only redirects `STDOUT`. To redirect `STDERR`, use `>`.

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(`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.

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(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` |> prefixer("A",2) & prefixer("B",2))
A  0
B  1
A  2
B  3
A  4
B  5
A  6
B  7
A  8
B  9
```

This is a classic example of a single producer feeding two concurrent consumers: one `perl` process generates lines with the numbers 0 through 9 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(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` |>
    prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3) |>
    prefixer("A",2) & prefixer("B",2))
B  Y  0
A  Z  1
B  X  2
A  Y  3
B  Z  4
A  X  5
B  Y  6
A  Z  7
B  X  8
A  Y  9
```

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.

1.23 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 by reference.

Finally, you can use `ccall` to actually generate a call to the library function. Arguments to `ccall` are as follows:

1. `(:function, "library")` pair (must be a constant, but see below).
2. Return type, which may be any bits type, including `Int32`, `Int64`, `Float64`, or `Ptr{T}` for any type parameter `T`, indicating a pointer to values of type `T`, or `Ptr{Void}` for `void*` “untyped pointer” values.
3. A tuple of input types, like those allowed for the return type. The input types must be written as a literal tuple, not a tuple-valued variable or expression.
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> 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> path = ccall( (:getenv, "libc"), Ptr{UInt8}, (Ptr{UInt8},), "SHELL")
Ptr{UInt8} @0x00007fff5fbffc45
```

```
julia> bytestring(path)
"/bin/bash"
```

Note that the argument type tuple must be written as `(Ptr{UInt8},)`, rather than `(Ptr{UInt8})`. This is because `(Ptr{UInt8})` is just `Ptr{UInt8}`, rather than a 1-tuple containing `Ptr{UInt8}`:

```
julia> (Ptr{UInt8})
Ptr{UInt8}

julia> (Ptr{UInt8},)
(Ptr{UInt8},)
```

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 in `env.jl`:

```
function getenv(var::String)
    val = ccall( (:getenv, "libc"),
                  Ptr{UInt8}, (Ptr{UInt8},), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    bytestring(val)
end
```

The C `getenv` function indicates an error by returning `NULL`, but other standard C functions indicate errors in various different ways, including by returning `-1`, `0`, `1` and other special values. This wrapper throws an exception clearly indicating the problem if the caller tries to get a non-existent environment variable:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname:

```
function gethostname()
    hostname = Array{UInt8, 128}
    ccall( (:gethostname, "libc"), Int32,
           (Ptr{UInt8}, UInt),
           hostname, length(hostname))
    return bytestring(convert{Ptr{UInt8}, 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.

A prefix `&` is used 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.

```
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,
                     (Ptr{Int32}, Ptr{Float64}, Ptr{Int32}, Ptr{Float64}, Ptr{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 `type`). However, even for immutable types 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)`.

Currently, it is not possible to reliably pass structs and other non-primitive types by *value* from Julia to/from C libraries. However, *pointers* to structs can be passed. The simplest case is that of C functions that generate and use *opaque* pointers to struct types, which can be passed to/from Julia as `Ptr{Void}` (or any other `Ptr` type). 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. A more complicated approach is to declare a composite type in Julia that mirrors a C struct, which allows the structure fields to be directly accessed in Julia. Given a Julia variable `x` of that type, a pointer can be passed as `&x` to a C function expecting a pointer to the corresponding struct. If the Julia type `T` is *immutable*, then a Julia `Array{T}` is stored in memory identically to a C array of the corresponding struct, and can be passed to a C program expecting such an array pointer.

Note that no C header files are used anywhere in the process: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file. (The *Clang* package <https://github.com/ihnorton/Clang.jl> can be used to generate Julia code from a C header file.)

1.23.1 Mapping C Types to Julia

Julia automatically inserts calls to the `convert` function to convert each argument to the specified type. For example, the following call:

```
ccall( (:foo, "libfoo"), Void, (Int32, Float64),
      x, y)
```

will behave as if the following were written:

```
ccall( (:foo, "libfoo"), Void, (Int32, Float64),
      convert{Int32}(x), convert{Float64}(y))
```

When a scalar value is passed with `&` as an argument of type `Ptr{T}`, the value will first be converted to type `T`.

Array conversions

When an array is passed to C as a `Ptr{T}` argument, it is never converted: Julia simply checks that the element type of the array matches `T`, and the address of the first element is passed. This is done in order to avoid copying arrays unnecessarily.

Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `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 either declare the argument as `Ptr{Void}` or you can explicitly call `convert{Ptr{T}}(pointer(A))`.

Type correspondences

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:

<code>signed char</code>		<code>Int8</code>
<code>unsigned char</code>	<code>Cuchar</code>	<code>UInt8</code>
<code>short</code>	<code>Cshort</code>	<code>Int16</code>
<code>unsigned short</code>	<code>Cushort</code>	<code>UInt16</code>
<code>int</code>	<code>Cint</code>	<code>Int32</code>
<code>unsigned int</code>	<code>Cuint</code>	<code>UInt32</code>
<code>long long</code>	<code>Clonglong</code>	<code>Int64</code>
<code>unsigned long long</code>	<code>Culonglong</code>	<code>UInt64</code>
<code>float</code>	<code>Cfloat</code>	<code>Float32</code>
<code>double</code>	<code>Cdouble</code>	<code>Float64</code>
<code>ptrdiff_t</code>	<code>Cptrdiff_t</code>	<code>Int</code>
<code>ssize_t</code>	<code>Cssize_t</code>	<code>Int</code>
<code>size_t</code>	<code>Csize_t</code>	<code>UInt</code>
<code>void</code>		<code>Void</code>
<code>void*</code>		<code>Ptr{Void}</code>
<code>char*</code> (or <code>char[]</code> , e.g. a string)		<code>Ptr{UInt8}</code>
<code>char**</code> (or <code>*char[]</code>)		<code>Ptr{Ptr{UInt8}}</code>
<code>struct T*</code> (where T represents an appropriately defined bits type)		<code>Ptr{T}</code> (call using <code>&variable_name</code> in the parameter list)
<code>j1_value_t*</code> (any Julia Type)		<code>Ptr{Any}</code>

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.

A C function declared to return `void` will give nothing in Julia.

System-dependent:

<code>char</code>	<code>Cchar</code>	<code>Int8</code> (x86, x86_64) <code>UInt8</code> (powerpc, arm)
<code>long</code>	<code>Clong</code>	<code>Int</code> (UNIX) <code>Int32</code> (Windows)
<code>unsigned long</code>	<code>Culong</code>	<code>UInt</code> (UNIX) <code>UInt32</code> (Windows)
<code>wchar_t</code>	<code>Cwchar_t</code>	<code>Int32</code> (UNIX) <code>UInt16</code> (Windows)

For string arguments (`char*`) the Julia type should be `Ptr{UInt8}`, not `ASCIIString`. 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)
```

For `wchar_t*` arguments, the Julia type should be `Ptr{Wchar_t}`, and data can be converted to/from ordinary Julia strings by the `wstring(s)` function (equivalent to either `utf16(s)` or `utf32(s)` depending upon the width of `Cwchar_t`. Note also that ASCII, UTF-8, UTF-16, and UTF-32 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy.

1.23.2 Accessing Data through a Pointer

The following methods are described as “unsafe” because they can cause Julia to terminate abruptly or corrupt arbitrary process memory due to a bad pointer or type declaration.

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 performs 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{Void}`, 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 bittypes or other pointer-free (`isbits`) immutable 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 (bitstype or immutable), the function `pointer_to_array(ptr, dims, [own])` 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.

1.23.3 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 encapsulate the value inside an array of the appropriate type. When you pass the array as an argument with a `Ptr` type, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Cint[0]
range = Cfloat[0]
ccall(:foo, Void, (Ptr{Cint}, Ptr{Cfloat}), width, range)
```

This is used extensively in Julia’s LAPACK interface, where an integer `info` is passed to LAPACK by reference, and on return, includes the success code.

1.23.4 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{Any, 1}` to hold these values, until C interface 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 `bytestring()` 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 `pointer_to_array()` 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.

1.23.5 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.

1.23.6 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 or function arguments.

1.23.7 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`. For example (from `base/libc.jl`):

```
hn = Array{UInt8, 256}
err=ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))
```

For more information, please see the [LLVM Language Reference](#).

1.23.8 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store`.

1.23.9 Passing Julia Callback Functions to C

It is possible to pass Julia functions to native functions that accept function pointer arguments. 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 *a, const void *b));
```

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 type `T`:

```
function mycompare{T}(a_::Ptr{T}, b_::Ptr{T})
    a = unsafe_load(a_)
    b = unsafe_load(b_)
    return convert{Cint, a < b ? -1 : a > b ? +1 : 0}
end
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a C `int`, so we must be sure to return `Cint` via a call to `convert`.

In order to pass this function to C, we obtain its address using the function `cfunction`:

```
const mycompare_c = cfunction(mycompare, Cint, (Ptr{Cdouble}, Ptr{Cdouble}))
```

`cfunction` accepts three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a tuple of the argument types, in this case to sort an array of `Cdouble` (`Float64`) elements.

The final call to `qsort` looks like this:

```
A = [1.3, -2.7, 4.4, 3.1]
ccall(:qsort, Void, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Void}),
      A, length(A), sizeof(eltype(A)), mycompare_c)
```

After this executes, `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).

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, you pass a function of one argument (the `AsyncWork` object for which the event was triggered, which you'll probably just ignore) to `SingleAsyncWork`:

```
cb = Base.SingleAsyncWork(data -> my_real_callback(args))
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cb.handle` as the argument.

More About Callbacks

For more details on how to pass callbacks to C libraries, see this [blog post](#).

1.23.10 C++

Limited support for C++ is provided by the `Cpp` and `Clang` packages.

1.23.11 Handling Platform Variations

When dealing with platform libraries, it is often necessary to provide special cases for various platforms. The variable `OS_NAME` can be used to write these special cases. Additionally, there are several macros intended to make this easier: `@windows`, `@unix`, `@linux`, and `@osx`. Note that `linux` and `osx` are mutually exclusive subsets of `unix`. Their usage takes the form of a ternary conditional operator, as demonstrated in the following examples.

Simple blocks:

```
ccall( (@windows? :_fopen : :fopen), ...)
```

Complex blocks:

```
@linux? (
    begin
        some_complicated_thing(a)
    end
: begin
    some_different_thing(a)
end
)
```

Chaining (parentheses optional, but recommended for readability):

```
@windows? :a : (@osx? :b : :c)
```

1.24 Embedding Julia

As we have seen (*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#).

1.24.1 High-Level Embedding

We start with a simple C program that initializes Julia and calls some Julia code:

```
#include <julia.h>

int main(int argc, char *argv[])
{
    jl_init(NULL);
    JL_SET_STACK_BASE;

    jl_eval_string("print(sqrt(2.0))");

    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 -I$JULIA_DIR/include/julia -L$JULIA_DIR/usr/lib -ljulia test.c
```

Alternatively, look at the `embedding.c` program in the `julia` source tree in the `examples/` folder.

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 takes as argument a C string (`const char*`) to the location where Julia is installed. When the argument is `NULL`, Julia tries to determine the install location automatically.

The second statement initializes Julia's task scheduling system. This statement must appear in a function that will not return as long as calls into Julia will be made (`main` works fine). Strictly speaking, this statement is optional, but operations that switch tasks will cause problems if it is omitted.

The third statement in the test program evaluates a Julia statement using a call to `jl_eval_string`.

1.24.2 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_is_float64(ret)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `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.

1.24.3 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`,

`j1_call12`, and `j1_call13` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `j1_call`:

```
j1_value_t *j1_call(j1_function_t *f, j1_value_t **args, int32_t nargs)
```

Its second argument `args` is an array of `j1_value_t*` arguments and `nargs` is the number of arguments.

1.24.4 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 `j1_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 `j1...` calls. But in order to make sure that values can survive `j1...` 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:

```
j1_value_t *ret = j1_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:

```
j1_value_t **args;
JL_GC_PUSHARGS(args, 2); // args can now hold 2 'j1_value_t*' objects
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call j1... functions)
JL_GC_POP();
```

Manipulating the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

<code>void j1_gc_collect()</code>	Force a GC run
<code>void j1_gc_disable()</code>	Disable the GC
<code>void j1_gc_enable()</code>	Enable the GC

1.24.5 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 `j1_array_t*`. Basically, `j1_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_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.

1.24.6 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_is_float64(val)) {
    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.

1.25 Packages

Julia has a built-in package manager for installing add-on functionality written in Julia. It can also install external libraries using your operating system's standard system for doing so, or by compiling from source. The list of registered Julia packages can be found at <http://pkg.julialang.org>. All package manager commands are found in the `Pkg` module, included in Julia's `Base` install.

1.25.1 Package Status

The `Pkg.status()` function prints out a summary of the state of packages you have installed. Initially, you'll have no packages installed:

```
julia> Pkg.status()
INFO: Initializing package repository /Users/stefan/.julia/v0.3
INFO: Cloning METADATA from git://github.com/JuliaLang/METADATA.jl
No packages installed.
```

Your package directory is automatically initialized the first time you run a `Pkg` command that expects it to exist – which includes `Pkg.status()`. Here's an example non-trivial set of required and additional packages:

```
julia> Pkg.status()
Required packages:
- Distributions          0.2.8
- UTF16                 0.2.0
Additional packages:
- NumericExtensions     0.2.17
- Stats                 0.2.6
```

These packages are all on registered versions, managed by `Pkg`. Packages can be in more complicated states, indicated by annotations to the right of the installed package version; we will explain these states and annotations as we encounter them. For programmatic usage, `Pkg.installed()` returns a dictionary, mapping installed package names to the version of that package which is installed:

```
julia> Pkg.installed()
["Distributions"=>v"0.2.8", "Stats"=>v"0.2.6", "UTF16"=>v"0.2.0", "NumericExtensions"=>v"0.2.17"]
```

1.25.2 Adding and Removing Packages

Julia's package manager is a little unusual in that it is declarative rather than imperative. This means that you tell it what you want and it figures out what versions to install (or remove) to satisfy those requirements optimally – and minimally. So rather than installing a package, you just add it to the list of requirements and then “resolve” what needs to be installed. In particular, this means that if some package had been installed because it was needed by a previous version of something you wanted, and a newer version doesn't have that requirement anymore, updating will actually remove that package.

Your package requirements are in the file `~/.julia/v0.3/REQUIRE`. You can edit this file by hand and then call `Pkg.resolve()` to install, upgrade or remove packages to optimally satisfy the requirements, or you can do `Pkg.edit()`, which will open `REQUIRE` in your editor (configured via the `EDITOR` or `VISUAL` environment variables), and then automatically call `Pkg.resolve()` afterwards if necessary. If you only want to add or remove the requirement for a single package, you can also use the non-interactive `Pkg.add()` and `Pkg.rm()` commands, which add or remove a single requirement to `REQUIRE` and then call `Pkg.resolve()`.

You can add a package to the list of requirements with the `Pkg.add()` function, and the package and all the packages that it depends on will be installed:

```
julia> Pkg.status()
No packages installed.

julia> Pkg.add("Distributions")
INFO: Cloning cache of Distributions from git://github.com/JuliaStats/Distributions.jl.git
INFO: Cloning cache of NumericExtensions from git://github.com/lindahua/NumericExtensions.jl.git
INFO: Cloning cache of Stats from git://github.com/JuliaStats/Stats.jl.git
INFO: Installing Distributions v0.2.7
INFO: Installing NumericExtensions v0.2.17
```



```
INFO: Installing Stats v0.2.6
INFO: REQUIRE updated.
```

```
julia> Pkg.status()
Required packages:
- Distributions                0.2.7
Additional packages:
- NumericExtensions           0.2.17
- Stats                        0.2.6
```

What this is doing is first adding `Distributions` to your `~/.julia/v0.3/REQUIRE` file:

```
$ cat ~/.julia/v0.3/REQUIRE
Distributions
```

It then runs `Pkg.resolve()` using these new requirements, which leads to the conclusion that the `Distributions` package should be installed since it is required but not installed. As stated before, you can accomplish the same thing by editing your `~/.julia/v0.3/REQUIRE` file by hand and then running `Pkg.resolve()` yourself:

```
$ echo UTF16 >> ~/.julia/v0.3/REQUIRE

julia> Pkg.resolve()
INFO: Cloning cache of UTF16 from git://github.com/nolta/UTF16.jl.git
INFO: Installing UTF16 v0.2.0

julia> Pkg.status()
Required packages:
- Distributions                0.2.7
- UTF16                        0.2.0
Additional packages:
- NumericExtensions           0.2.17
- Stats                        0.2.6
```

This is functionally equivalent to calling `Pkg.add("UTF16")`, except that `Pkg.add()` doesn't change `REQUIRE` until *after* installation has completed, so if there are problems, `REQUIRE` will be left as it was before calling `Pkg.add()`. The format of the `REQUIRE` file is described in [Requirements Specification](#); it allows, among other things, requiring specific ranges of versions of packages.

When you decide that you don't want to have a package around any more, you can use `Pkg.rm()` to remove the requirement for it from the `REQUIRE` file:

```
julia> Pkg.rm("Distributions")
INFO: Removing Distributions v0.2.7
INFO: Removing Stats v0.2.6
INFO: Removing NumericExtensions v0.2.17
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
- UTF16                        0.2.0

julia> Pkg.rm("UTF16")
INFO: Removing UTF16 v0.2.0
INFO: REQUIRE updated.

julia> Pkg.status()
No packages installed.
```

Once again, this is equivalent to editing the `REQUIRE` file to remove the line with each package name on it then running `Pkg.resolve()` to update the set of installed packages to match. While `Pkg.add()` and `Pkg.rm()` are convenient for adding and removing requirements for a single package, when you want to add or remove multiple packages, you can call `Pkg.edit()` to manually change the contents of `REQUIRE` and then update your packages accordingly. `Pkg.edit()` does not roll back the contents of `REQUIRE` if `Pkg.resolve()` fails – rather, you have to run `Pkg.edit()` again to fix the files contents yourself.

Because the package manager uses git internally to manage the package git repositories, users may run into protocol issues (if behind a firewall, for example), when running `Pkg.add()`. The following command can be run from the command line to tell git to use ‘https’ instead of the ‘git’ protocol when cloning repositories:

```
git config --global url."https://".insteadOf git://
```

1.25.3 Installing Unregistered Packages

Julia packages are simply git repositories, clonable via any of the [protocols](#) that git supports, and containing Julia code that follows certain layout conventions. Official Julia packages are registered in the `METADATA.jl` repository, available at a well-known location ⁴. The `Pkg.add()` and `Pkg.rm()` commands in the previous section interact with registered packages, but the package manager can install and work with unregistered packages too. To install an unregistered package, use `Pkg.clone(url)`, where `url` is a git URL from which the package can be cloned:

```
julia> Pkg.clone("git://example.com/path/to/Package.jl.git")
INFO: Cloning Package from git://example.com/path/to/Package.jl.git
Cloning into 'Package'...
remote: Counting objects: 22, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 22 (delta 8), reused 22 (delta 8)
Receiving objects: 100% (22/22), 2.64 KiB, done.
Resolving deltas: 100% (8/8), done.
```

By convention, Julia repository names end with `.jl` (the additional `.git` indicates a “bare” git repository), which keeps them from colliding with repositories for other languages, and also makes Julia packages easy to find in search engines. When packages are installed in your `.julia/v0.3` directory, however, the extension is redundant so we leave it off.

If unregistered packages contain a `REQUIRE` file at the top of their source tree, that file will be used to determine which registered packages the unregistered package depends on, and they will automatically be installed. Unregistered packages participate in the same version resolution logic as registered packages, so installed package versions will be adjusted as necessary to satisfy the requirements of both registered and unregistered packages.

1.25.4 Updating Packages

When package developers publish new registered versions of packages that you’re using, you will, of course, want the new shiny versions. To get the latest and greatest versions of all your packages, just do `Pkg.update()`:

```
julia> Pkg.update()
INFO: Updating METADATA...
INFO: Computing changes...
INFO: Upgrading Distributions: v0.2.8 => v0.2.10
INFO: Upgrading Stats: v0.2.7 => v0.2.8
```

⁴ The official set of packages is at <https://github.com/JuliaLang/METADATA.jl>, but individuals and organizations can easily use a different metadata repository. This allows control which packages are available for automatic installation. One can allow only audited and approved package versions, and make private packages or forks available.

The first step of updating packages is to pull new changes to `~/.julia/v0.3/METADATA` and see if any new registered package versions have been published. After this, `Pkg.update()` attempts to update packages that are checked out on a branch and not dirty (i.e. no changes have been made to files tracked by git) by pulling changes from the package’s upstream repository. Upstream changes will only be applied if no merging or rebasing is necessary – i.e. if the branch can be “fast-forwarded”. If the branch cannot be fast-forwarded, it is assumed that you’re working on it and will update the repository yourself.

Finally, the update process recomputes an optimal set of package versions to have installed to satisfy your top-level requirements and the requirements of “fixed” packages. A package is considered fixed if it is one of the following:

1. **Unregistered:** the package is not in METADATA – you installed it with `Pkg.clone()`.
2. **Checked out:** the package repo is on a development branch.
3. **Dirty:** changes have been made to files in the repo.

If any of these are the case, the package manager cannot freely change the installed version of the package, so its requirements must be satisfied by whatever other package versions it picks. The combination of top-level requirements in `~/.julia/v0.3/REQUIRE` and the requirement of fixed packages are used to determine what should be installed.

1.25.5 Checkout, Pin and Free

You may want to use the `master` version of a package rather than one of its registered versions. There might be fixes or functionality on `master` that you need that aren’t yet published in any registered versions, or you may be a developer of the package and need to make changes on `master` or some other development branch. In such cases, you can do `Pkg.checkout(pkg)` to checkout the `master` branch of `pkg` or `Pkg.checkout(pkg, branch)` to checkout some other branch:

```
julia> Pkg.add("Distributions")
INFO: Installing Distributions v0.2.9
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.7
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
- Distributions                0.2.9
Additional packages:
- NumericExtensions          0.2.17
- Stats                      0.2.7

julia> Pkg.checkout("Distributions")
INFO: Checking out Distributions master...
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
- Distributions                0.2.9+          master
Additional packages:
- NumericExtensions          0.2.17
- Stats                      0.2.7
```

Immediately after installing `Distributions` with `Pkg.add()` it is on the current most recent registered version – `0.2.9` at the time of writing this. Then after running `Pkg.checkout("Distributions")`, you can see from the output of `Pkg.status()` that `Distributions` is on an unregistered version greater than `0.2.9`, indicated by the “pseudo-version” number `0.2.9+`.

When you checkout an unregistered version of a package, the copy of the `REQUIRE` file in the package repo takes precedence over any requirements registered in `METADATA`, so it is important that developers keep this file accurate and up-to-date, reflecting the actual requirements of the current version of the package. If the `REQUIRE` file in the package repo is incorrect or missing, dependencies may be removed when the package is checked out. This file is also used to populate newly published versions of the package if you use the API that `Pkg` provides for this (described below).

When you decide that you no longer want to have a package checked out on a branch, you can “free” it back to the control of the package manager with `Pkg.free(pkg)`:

```
julia> Pkg.free("Distributions")
INFO: Freeing Distributions...
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
- Distributions              0.2.9
Additional packages:
- NumericExtensions         0.2.17
- Stats                     0.2.7
```

After this, since the package is on a registered version and not on a branch, its version will be updated as new registered versions of the package are published.

If you want to pin a package at a specific version so that calling `Pkg.update()` won’t change the version the package is on, you can use the `Pkg.pin()` function:

```
julia> Pkg.pin("Stats")
INFO: Creating Stats branch pinned.47c198b1.tmp

julia> Pkg.status()
Required packages:
- Distributions              0.2.9
Additional packages:
- NumericExtensions         0.2.17
- Stats                     0.2.7                pinned.47c198b1.tmp
```

After this, the `Stats` package will remain pinned at version `0.2.7` – or more specifically, at commit `47c198b1`, but since versions are permanently associated a given git hash, this is the same thing. `Pkg.pin()` works by creating a throw-away branch for the commit you want to pin the package at and then checking that branch out. By default, it pins a package at the current commit, but you can choose a different version by passing a second argument:

```
julia> Pkg.pin("Stats",v"0.2.5")
INFO: Creating Stats branch pinned.1fd0983b.tmp
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
- Distributions              0.2.9
Additional packages:
- NumericExtensions         0.2.17
- Stats                     0.2.5                pinned.1fd0983b.tmp
```

Now the `Stats` package is pinned at commit `1fd0983b`, which corresponds to version `0.2.5`. When you decide to “unpin” a package and let the package manager update it again, you can use `Pkg.free()` like you would to move off of any branch:

```
julia> Pkg.free("Stats")
INFO: Freeing Stats...
```

```
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
- Distributions              0.2.9
Additional packages:
- NumericExtensions         0.2.17
- Stats                      0.2.7
```

After this, the `Stats` package is managed by the package manager again, and future calls to `Pkg.update()` will upgrade it to newer versions when they are published. The throw-away `pinned.1fd0983b.tmp` branch remains in your local `Stats` repo, but since git branches are extremely lightweight, this doesn't really matter; if you feel like cleaning them up, you can go into the repo and delete those branches.

1.26 Package Development

Julia's package manager is designed so that when you have a package installed, you are already in a position to look at its source code and full development history. You are also able to make changes to packages, commit them using git, and easily contribute fixes and enhancements upstream. Similarly, the system is designed so that if you want to create a new package, the simplest way to do so is within the infrastructure provided by the package manager.

1.26.1 Initial Setup

Since packages are git repositories, before doing any package development you should setup the following standard global git configuration settings:

```
$ git config --global user.name "FULL NAME"
$ git config --global user.email "EMAIL"
```

where `FULL NAME` is your actual full name (spaces are allowed between the double quotes) and `EMAIL` is your actual email address. Although it isn't necessary to use [GitHub](#) to create or publish Julia packages, most Julia packages as of writing this are hosted on GitHub and the package manager knows how to format origin URLs correctly and otherwise work with the service smoothly. We recommend that you create a [free account](#) on GitHub and then do:

```
$ git config --global github.user "USERNAME"
```

where `USERNAME` is your actual GitHub user name. Once you do this, the package manager knows your GitHub user name and can configure things accordingly. You should also [upload](#) your public SSH key to GitHub and set up an [SSH agent](#) on your development machine so that you can push changes with minimal hassle. In the future, we will make this system extensible and support other common git hosting options like [BitBucket](#) and allow developers to choose their favorite.

1.26.2 Guidelines for Naming a Package

Package names should be sensible to most Julia users, *even to those who are not domain experts*.

1. Avoid jargon. In particular, avoid acronyms unless there is minimal possibility of confusion.
 - It's ok to say `USA` if you're talking about the USA.
 - It's not ok to say `PMA`, even if you're talking about positive mental attitude.
2. Packages that provide most of their functionality in association with a new type should have pluralized names.
 - `DataFrames` provides the `DataFrame` type.

- `BloomFilters` provides the `BloomFilter` type.
 - In contrast, `JuliaParser` provides no new type, but instead new functionality in the `JuliaParser.parse()` function.
3. Err on the side of clarity, even if clarity seems long-winded to you.
 - `RandomMatrices` is a less ambiguous name than `RndMat` or `RMT`, even though the latter are shorter.
 4. A less systematic name may suit a package that implements one of several possible approaches to its domain.
 - Julia does not have a single comprehensive plotting package. Instead, `Gadfly`, `PyPlot`, `Winston` and other packages each implement a unique approach based on a particular design philosophy.
 - In contrast, `SortingAlgorithms` provides a consistent interface to use many well-established sorting algorithms.
 5. Packages that wrap external libraries or programs should be named after those libraries or programs.
 - `CPLEX.jl` wraps the `CPLEX` library, which can be identified easily in a web search.
 - `MATLAB.jl` provides an interface to call the `MATLAB` engine from within Julia.

1.26.3 Generating a New Package

Suppose you want to create a new Julia package called `FooBar`. To get started, do `Pkg.generate(pkg, license)` where `pkg` is the new package name and `license` is the name of a license that the package generator knows about:

```
julia> Pkg.generate("FooBar", "MIT")
INFO: Initializing FooBar repo: /Users/stefan/.julia/v0.3/FooBar
INFO: Origin: git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Generating LICENSE.md
INFO: Generating README.md
INFO: Generating src/FooBar.jl
INFO: Generating test/runtests.jl
INFO: Generating .travis.yml
INFO: Committing FooBar generated files
```

This creates the directory `~/.julia/v0.3/FooBar`, initializes it as a git repository, generates a bunch of files that all packages should have, and commits them to the repository:

```
$ cd ~/.julia/v0.3/FooBar && git show --stat

commit 84b8e266dae6de30ab9703150b3bf771ec7b6285
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 17:57:58 2013 -0400

    FooBar.jl generated files.

    license: MIT
    authors: Stefan Karpinski
    years:   2013
    user:    StefanKarpinski

Julia Version 0.3.0-prerelease+3217 [5fcfb13*]

.travis.yml      | 16 +++++
LICENSE.md       | 22 +++++
README.md        |  3 +++
```

```
src/FooBar.jl      |  5 +++++
test/runtests.jl  |  5 +++++
5 files changed, 51 insertions(+)
```

At the moment, the package manager knows about the MIT “Expat” License, indicated by "MIT", the Simplified BSD License, indicated by "BSD", and version 2.0 of the Apache Software License, indicated by "ASL". If you want to use a different license, you can ask us to add it to the package generator, or just pick one of these three and then modify the `~/.julia/v0.3/PACKAGE/LICENSE.md` file after it has been generated.

If you created a GitHub account and configured git to know about it, `Pkg.generate()` will set an appropriate origin URL for you. It will also automatically generate a `.travis.yml` file for using the Travis automated testing service. You will have to enable testing on the Travis website for your package repository, but once you’ve done that, it will already have working tests. Of course, all the default testing does is verify that using `FooBar` in Julia works.

1.26.4 Making Your Package Available

Once you’ve made some commits and you’re happy with how `FooBar` is working, you may want to get some other people to try it out. First you’ll need to create the remote repository and push your code to it; we don’t yet automatically do this for you, but we will in the future and it’s not too hard to figure out⁵. Once you’ve done this, letting people try out your code is as simple as sending them the URL of the published repo – in this case:

```
git://github.com/StefanKarpinski/FooBar.jl.git
```

For your package, it will be your GitHub user name and the name of your package, but you get the idea. People you send this URL to can use `Pkg.clone()` to install the package and try it out:

```
julia> Pkg.clone("git://github.com/StefanKarpinski/FooBar.jl.git")
INFO: Cloning FooBar from git@github.com:StefanKarpinski/FooBar.jl.git
```

1.26.5 Publishing Your Package

Once you’ve decided that `FooBar` is ready to be registered as an official package, you can add it to your local copy of METADATA using `Pkg.register()`:

```
julia> Pkg.register("FooBar")
INFO: Registering FooBar at git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Committing METADATA for FooBar
```

This creates a commit in the `~/.julia/v0.3/METADATA` repo:

```
$ cd ~/.julia/v0.3/METADATA && git show

commit 9f71f4becb05cadacb983c54a72eed744e5c019d
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 18:46:02 2013 -0400
```

```
    Register FooBar
```

```
diff --git a/FooBar/url b/FooBar/url
new file mode 100644
index 0000000..30e525e
--- /dev/null
+++ b/FooBar/url
```

⁵ Installing and using GitHub’s “hub” tool is highly recommended. It allows you to do things like `run hub create` in the package repo and have it automatically created via GitHub’s API.

```
@@ -0,0 +1 @@
+git://github.com/StefanKarpinski/FooBar.jl.git
```

This commit is only locally visible, however. In order to make it visible to the world, you need to merge your local METADATA upstream into the official repo. The `Pkg.publish()` command will fork the METADATA repository on GitHub, push your changes to your fork, and open a pull request:

```
julia> Pkg.publish()
INFO: Validating METADATA
INFO: No new package versions to publish
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/ef45f54b
INFO: To create a pull-request open:

https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/ef45f54b
```

Tip: If `Pkg.publish()` fails with error:

```
ERROR: key not found: "token"
```

then you may have encountered an issue from using the GitHub API on multiple systems. The solution is to delete the “Julia Package Manager” personal access token *from your GitHub account* and try again.

Other failures may require you to circumvent `Pkg.publish()` by creating a pull request on GitHub. See: *Publishing METADATA manually* below.

Once the package URL for `FooBar` is registered in the official METADATA repo, people know where to clone the package from, but there still aren’t any registered versions available. This means that `Pkg.add("FooBar")` won’t work yet since it only installs official versions. `Pkg.clone("FooBar")` without having to specify a URL for it. Moreover, when they run `Pkg.update()`, they will get the latest version of `FooBar` that you’ve pushed to the repo. This is a good way to have people test out your packages as you work on them, before they’re ready for an official release.

Publishing METADATA manually

If `Pkg.publish()` fails you can follow these instructions to manually publish your package.

By “forking” the main METADATA repository, you can create a personal copy (of `METADATA.jl`) under your GitHub account. Once that copy exists, you can push your local changes to your copy (just like any other GitHub project).

1. go to <https://github.com/JuliaLang/METADATA.jl/fork> and create your own fork.
2. add your fork as a remote repository for the METADATA repository on your local computer (in the terminal where USERNAME is your github username):

```
cd ~/.julia/METADATA
git remote add USERNAME https://github.com/USERNAME/METADATA.jl.git
```

3. push your changes to your fork:

```
git push USERNAME metadata-v2
```

4. If all of that works, then go back to the GitHub page for your fork, and click the “pull request” link.

1.26.6 Tagging Package Versions

Once you are ready to make an official version your package, you can tag and register it with the `Pkg.tag()` command:

```
julia> Pkg.tag("FooBar")
INFO: Tagging FooBar v0.0.1
INFO: Committing METADATA for FooBar
```

This tags `v0.0.1` in the `FooBar` repo:

```
$ cd ~/.julia/v0.3/FooBar && git tag
v0.0.1
```

It also creates a new version entry in your local METADATA repo for `FooBar`:

```
$ cd ~/.julia/v0.3/FooBar && git show
commit de77ee4dc0689b12c5e8b574aef7f70e8b311b0e
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 23:06:18 2013 -0400

    Tag FooBar v0.0.1

diff --git a/FooBar/versions/0.0.1/sha1 b/FooBar/versions/0.0.1/sha1
new file mode 100644
index 0000000..c1cb1c1
--- /dev/null
+++ b/FooBar/versions/0.0.1/sha1
@@ -0,0 +1 @@
+84b8e266dae6de30ab9703150b3bf771ec7b6285
```

If there is a `REQUIRE` file in your package repo, it will be copied into the appropriate spot in METADATA when you tag a version. Package developers should make sure that the `REQUIRE` file in their package correctly reflects the requirements of their package, which will automatically flow into the official metadata if you're using `Pkg.tag()`. See the [Requirements Specification](#) for the full format of `REQUIRE`.

The `Pkg.tag()` command takes an optional second argument that is either an explicit version number object like `v"0.0.1"` or one of the symbols `:patch`, `:minor` or `:major`. These increment the patch, minor or major version number of your package intelligently.

As with `Pkg.register()`, these changes to METADATA aren't available to anyone else until they've been included upstream. Again, use the `Pkg.publish()` command, which first makes sure that individual package repos have been tagged, pushes them if they haven't already been, and then opens a pull request to METADATA:

```
julia> Pkg.publish()
INFO: Validating METADATA
INFO: Pushing FooBar permanent tags: v0.0.1
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/3ef4f5c4
INFO: To create a pull-request open:

https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/3ef4f5c4
```

1.26.7 Fixing Package Requirements

If you need to fix the registered requirements of an already-published package version, you can do so just by editing the metadata for that version, which will still have the same commit hash – the hash associated with a version is

permanent:

```
$ cd ~/.julia/v0.3/METADATA/FooBar/versions/0.0.1 && cat requires
julia 0.3-
$ vi requires
```

Since the commit hash stays the same, the contents of the `REQUIRE` file that will be checked out in the repo will **not** match the requirements in `METADATA` after such a change; this is unavoidable. When you fix the requirements in `METADATA` for a previous version of a package, however, you should also fix the `REQUIRE` file in the current version of the package.

1.26.8 Requirements Specification

The `~/.julia/v0.3/REQUIRE` file, the `REQUIRE` file inside packages, and the `METADATA` package `requires` files use a simple line-based format to express the ranges of package versions which need to be installed. Package `REQUIRE` and `METADATA requires` files should also include the range of versions of `julia` the package is expected to work with.

Here's how these files are parsed and interpreted.

- Everything after a `#` mark is stripped from each line as a comment.
- If nothing but whitespace is left, the line is ignored.
- If there are non-whitespace characters remaining, the line is a requirement and the is split on whitespace into words.

The simplest possible requirement is just the name of a package name on a line by itself:

```
Distributions
```

This requirement is satisfied by any version of the `Distributions` package. The package name can be followed by zero or more version numbers in ascending order, indicating acceptable intervals of versions of that package. One version opens an interval, while the next closes it, and the next opens a new interval, and so on; if an odd number of version numbers are given, then arbitrarily large versions will satisfy; if an even number of version numbers are given, the last one is an upper limit on acceptable version numbers. For example, the line:

```
Distributions 0.1
```

is satisfied by any version of `Distributions` greater than or equal to `0.1.0`. Suffixing a version with `-` allows any pre-release versions as well. For example:

```
Distributions 0.1-
```

is satisfied by pre-release versions such as `0.1-dev` or `0.1-rc1`, or by any version greater than or equal to `0.1.0`.

This requirement entry:

```
Distributions 0.1 0.2.5
```

is satisfied by versions from `0.1.0` up to, but not including `0.2.5`. If you want to indicate that any `0.1.x` version will do, you will want to write:

```
Distributions 0.1 0.2-
```

If you want to start accepting versions after `0.2.7`, you can write:

```
Distributions 0.1 0.2- 0.2.7
```

If a requirement line has leading words that begin with `@`, it is a system-dependent requirement. If your system matches these system conditionals, the requirement is included, if not, the requirement is ignored. For example:

```
@osx Homebrew
```

will require the Homebrew package only on systems where the operating system is OS X. The system conditions that are currently supported are:

```
@windows
@unix
@osx
@linux
```

The `@unix` condition is satisfied on all UNIX systems, including OS X, Linux and FreeBSD. Negated system conditionals are also supported by adding a `!` after the leading `@`. Examples:

```
@!windows
@unix @!osx
```

The first condition applies to any system but Windows and the second condition applies to any UNIX system besides OS X.

Runtime checks for the current version of Julia can be made using the built-in `VERSION` variable, which is of type `VersionNumber`. Such code is occasionally necessary to keep track of new or deprecated functionality between various releases of Julia. Examples of runtime checks:

```
VERSION < v"0.3-" #exclude all pre-release versions of 0.3

v"0.2-" <= VERSION < v"0.3-" #get all 0.2 versions, including pre-releases, up to the above

v"0.2" <= VERSION < v"0.3-" #To get only stable 0.2 versions (Note v"0.2" == v"0.2.0")

VERSION >= v"0.2.1" #get at least version 0.2.1
```

See the section on *version number literals* for a more complete description.

1.27 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.

1.27.1 Basic usage

Let’s work with a simple test case:

```
function myfunc()
    A = rand(100, 100, 200)
    maximum(A)
end
```

It’s a good idea to first run the code you intend to profile at least once (unless you want to profile Julia’s JIT-compiler):

```
julia> myfunc()  # run once to force compilation
```

Now we’re ready to profile this function:

```
julia> @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()
 23 client.jl; _start; line: 373
 23 client.jl; run_repl; line: 166
 23 client.jl; eval_user_input; line: 91
 23 profile.jl; anonymous; line: 14
   8 none; myfunc; line: 2
   8 dsfmt.jl; dsfmt_gv_fill_array_close_open!; line: 128
 15 none; myfunc; line: 3
   2 reduce.jl; max; line: 35
   2 reduce.jl; max; line: 36
 11 reduce.jl; max; line: 37
```

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” indicates the number of backtraces (samples) taken *at this line or in any functions executed by this line*. The second field is the file name, followed by a semicolon; the third is the function name followed by a semicolon, and the fourth is the line number. 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 is `client.jl`’s `_start` function. This is the first Julia function that gets called when you launch julia. If you examine line 373 of `client.jl`, you’ll see that (at the time of this writing) it calls `run_repl`, mentioned on the second line. This in turn calls `eval_user_input`. These are the functions in `client.jl` that interpret 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 23 backtraces were taken at line 373 of `client.jl`, but it’s not that this line was “expensive” on its own: the second line reveals that all 23 of these backtraces were actually triggered inside its call to `run_repl`, 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:

```
8 none; myfunc; line: 2
```

`none` 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. Line 2 of `myfunc()` contains the call to `rand`, and there were 8 (out of 23) backtraces that occurred at this line. Below that, you can see a call to `dsfmt_gv_fill_array_close_open!` inside `dsfmt.jl`. You might be surprised not to see the `rand` function listed explicitly: that's because `rand` is *inlined*, and hence doesn't appear in the backtraces.

A little further down, you see:

```
15 none; myfunc; line: 3
```

Line 3 of `myfunc` contains the call to `max`, and there were 15 (out of 23) backtraces taken here. Below that, you can see the specific places in `base/reduce.jl` that carry out the time-consuming operations in the `max` function for this type of input data.

Overall, we can tentatively conclude that finding the maximum element is approximately twice as expensive as generating the random numbers. We could increase our confidence in this result by collecting more samples:

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
3121 client.jl; _start; line: 373
3121 client.jl; run_repl; line: 166
3121 client.jl; eval_user_input; line: 91
3121 profile.jl; anonymous; line: 1
848 none; myfunc; line: 2
842 dsfmt.jl; dsfmt_gv_fill_array_close_open!; line: 128
1510 none; myfunc; line: 3
74 reduce.jl; max; line: 35
122 reduce.jl; max; line: 36
1314 reduce.jl; max; line: 37
```

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](#).)

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      Function      Line
3121 client.jl  _start       373
3121 client.jl  eval_user_input  91
3121 client.jl  run_repl      166
842 dsfmt.jl   dsfmt_gv_fill_array_close_open! 128
848 none       myfunc        2
1510 none      myfunc        3
3121 profile.jl anonymous      1
74 reduce.jl   max           35
122 reduce.jl   max           36
1314 reduce.jl   max           37
```

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.

1.27.2 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()`.

1.27.3 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(); format = :tree, C = false, combine = true, cols = tt
```

Let’s discuss these arguments in order:

- The first argument allows you to save the results to a file, but the default is to print to `STDOUT` (the console).
- The second argument 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 first keyword argument, `format`, was introduced above. The possible choices are `:tree` and `:flat`.
- `C`, if set to `true`, allows you to see even the calls to C code. 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.
- 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`.
- `cols` allows you to control the number of columns that you are willing to use for display. When the text would be wider than the display, you might see output like this:

```
33 inference.jl; abstract_call; line: 645
33 inference.jl; abstract_call; line: 645
33 ..ence.jl; abstract_call_gf; line: 567
33 ..nce.jl; typeinf; line: 1201
+1 5 ..nce.jl; ..t_interpret; line: 900
+3 5 ..ence.jl; abstract_eval; line: 758
+4 5 ..ence.jl; ..ct_eval_call; line: 733
+6 5 ..ence.jl; abstract_call; line: 645
```

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 and use a very wide `cols` setting:

```
s = open("/tmp/prof.txt", "w")
Profile.print(s, cols = 500)
close(s)
```

1.27.4 Configuration

`@profile` just accumulates backtraces, and the analysis happens when you call `Profile.print()`. For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

```
Profile.init()           # returns the current settings
Profile.init(n, delay)
Profile.init(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).

1.28 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 `clear_malloc_data()` to reset all allocation counters. Finally, execute the desired commands and quit julia to trigger the generation of the `.mem` files.

1.29 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.

1.29.1 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
y = f(x::Int + 1)
```

Writing functions is better style. It leads to more reusable code and clarifies what steps are being done, and what their 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.

In the following REPL session:

```
julia> x = 1.0
```

is equivalent to:

```
julia> global x = 1.0
```

so all the performance issues discussed previously apply.

1.29.2 Measure performance with `@time` and pay attention to memory allocation

The most useful tool for measuring performance is the `@time` macro. The following example illustrates good working style:

```
julia> function f(n)
    s = 0
    for i = 1:n
        s += i/2
    end
    s
end
f (generic function with 1 method)

julia> @time f(1)
elapsed time: 0.008217942 seconds (93784 bytes allocated)
0.5

julia> @time f(10^6)
```



```
elapsed time: 0.063418472 seconds (32002136 bytes allocated)
2.5000025e11
```

On the first call (`@time f(1)`), `f` 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 large amount of memory was allocated. This is the single biggest advantage of `@time` vs. functions like `tic()` and `toc()`, which only report time.

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability. 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.

As a teaser, note that an improved version of this function allocates no memory (except to pass back the result back to the REPL) and has thirty-fold faster execution:

```
julia> @time f_improved(10^6)
elapsed time: 0.00253829 seconds (112 bytes allocated)
2.5000025e11
```

Below you’ll learn how to spot the problem with `f` and how to fix it.

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](#)).

1.29.3 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.
- 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.
- The [TypeCheck](#) package can help identify certain kinds of type problems. A more laborious but comprehensive tool is `code_typed()`. Look particularly for variables that have type `Any` (in the header) or statements declared as `Union` types. Such problems can usually be fixed using the tips below.
- The [Lint](#) package can also warn you of certain types of programming errors.

1.29.4 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:

```
a = Real[]      # typeof(a) = Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

Because `a` is 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. Because `f` will always be a `Float64`, we should instead, use:

```
a = Float64[] # typeof(a) = Array{Float64,1}
```

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under *Parametric Types*.

1.29.5 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.

Declare specific types for fields of composite types

Given a user-defined type like the following:

```
type Foo
    field
end
```

the compiler will not generally know the type of `foo.field`, since it might be modified at any time to refer to a value of a different type. It will help to declare the most specific type possible, such as `field::Float64` or `field::Array{Int64,1}`.

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type, such as the original `Foo` type above, or cell arrays (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.

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.

1.29.6 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:

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return max(svd(A)[2])
    else
        error("norm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
norm(x::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = max(svd(A)[2])
```

1.29.7 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 `one()` function, and a more general `oftype(x,y)` function, which returns `y` converted to the type of `x`. The first argument to any of these functions can be either a value or a type.

1.29.8 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 = x/bar()
    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 = one{T}`

1.29.9 Separate kernel functions

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:

```
function strange_twos(n)
    a = Array{randbool() ? Int64 : Float64, n}
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

This should be written as:

```
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
    end
end

function strange_twos(n)
    a = Array{randbool() ? Int64 : Float64, n}
    fill_twos!(a)
    return a
end
```

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 the standard library. For example, see `hvcats_fill` 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.

1.29.10 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]
2x2 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 `repmat()`):

```
function copy_cols{T}(x::Vector{T})
    n = size(x, 1)
    out = Array{eltype(x), n, n}
    for i=1:n
        out[:, i] = x
    end
    out
end

function copy_rows{T}(x::Vector{T})
    n = size(x, 1)
    out = Array{eltype(x), n, n}
    for i=1:n
        out[i, :] = x
    end
    out
end

function copy_col_row{T}(x::Vector{T})
    n = size(x, 1)
    out = Array{T, n, n}
    for col=1:n, row=1:n
        out[row, col] = x[row]
    end
    out
end

function copy_row_col{T}(x::Vector{T})
    n = size(x, 1)
    out = Array{T, n, n}
    for row=1:n, col=1:n
        out[row, col] = x[col]
    end
end
```

```
        out
    end
```

Now we will time each of these functions using the same random 10000 by 1 input vector:

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, {copy_cols, copy_rows, copy_col_row, copy_row_col});
copy_cols:      0.331706323
copy_rows:      1.799009911
copy_col_row:   0.415630047
copy_row_col:   1.721531501
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

1.29.11 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

```
function xinc(x)
    return [x, x+1, x+2]
end
```

```
function loopinc()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
    y
end
```

with

```
function xinc!{T}(ret::AbstractVector{T}, x::T)
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end
```

```
function loopinc_prealloc()
    ret = Array{Int, 3}
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
        y += ret[2]
    end
end
```

```

        y
    end

```

Timing results:

```

julia> @time loopinc()
elapsed time: 1.955026528 seconds (1279975584 bytes allocated)
50000015000000

julia> @time loopinc_prealloc()
elapsed time: 0.078639163 seconds (144 bytes allocated)
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.

1.29.12 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))
```

1.29.13 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.

1.29.14 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 `*` instead of raising to small integer powers, for example `x*x*x` instead of `x^3`.
- 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)`, and `fld(x,y)` instead of `floor(x/y)`.

1.29.15 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.
- Write `@simd` in front of `for` loops that are amenable to vectorization. **This feature is experimental** and could change or disappear in future versions of Julia.

Here is an example with both forms of markup:

```
function inner( x, y )
    s = zero(eltype(x))
    for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end

function innersimd( x, y )
    s = zero(eltype(x))
    @simd for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    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          = ",2.0*n*reps/time*1E-9)
    time = @elapsed for j in 1:reps
        s+=innersimd(x,y)
    end
    println("GFlop (SIMD) = ",2.0*n*reps/time*1E-9)
end

timeit(1000,1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
GFlop          = 1.9467069505224963
GFlop (SIMD) = 17.578554163920018
```

The range for a `@simd for` loop should be a one-dimensional range. A variable used for accumulating, such as `s` in the example, is called a *reduction variable*. 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`.
- No iteration ever waits on another iteration to make forward progress.

A `break`, `continue`, or `:obj'@goto'` in an `@simd` loop may cause wrong results.

Using `@simd` merely gives the compiler license to vectorize. Whether it actually does so depends on the compiler. To actually benefit from the current implementation, your loop should have the following additional properties:

- The loop must be an innermost loop.
- The loop body must be straight-line code. This is why `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `?:` expressions into straight-line code, if it is safe to evaluate all operands unconditionally. Consider using `ifelse()` 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.
- In some simple cases, for example with 2-3 arrays accessed in a loop, the LLVM auto-vectorization may kick in automatically, leading to no further speedup with `@simd`.

1.30 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.

1.30.1 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`).

1.30.2 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
convert(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 + one(x) # any integer type
addone(x::Number) = x + one(x)  # any numeric type
addone(x) = x + one(x)          # any type supporting + and one
```

The last definition of `addone` handles any type supporting `one()` (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 + one(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 `one()` replaced by its inlined value 1. Therefore, the first three definitions of `addone` above are completely redundant.

1.30.3 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.

1.30.4 Append ! to names of functions that modify their arguments

Instead of:

```
function double{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
    a
end
```

use:

```
function double!{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
    a
end
```

The Julia standard library 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.

1.30.5 Avoid strange type Unions

Types such as `Union{Function, String}` are often a sign that some design could be cleaner.

1.30.6 Try to avoid nullable fields

When using `x::Union{Nothing, T}`, ask whether the option for `x` to be `nothing` is really necessary. Here are some alternatives to consider:

- Find a safe default value to initialize `x` with
- Introduce another type that lacks `x`
- If there are many fields like `x`, store them in a dictionary
- Determine whether there is a simple rule for when `x` is `nothing`. For example, often the field will start as `nothing` but get initialized at some well-defined point. In that case, consider leaving it undefined at first.

1.30.7 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
a = Array{Union{Int, String, Tuple, Array}, n}
```

In this case `cell{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.

1.30.8 Use naming conventions consistent with Julia's base/

- modules and type names use capitalization and camel case: `module SparseMatrix`, `immutable 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 `remotecall(fetch(...))`) or as modifiers (`sum_kbn()`).
- 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.

1.30.9 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

1.30.10 Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

1.30.11 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.

1.30.12 Don't use unnecessary static parameters

A function signature:

```
foo{T<:Real}(x::T) = ...
```

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 *How should I declare “abstract container type” fields?* for more information.

1.30.13 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`) 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.

1.30.14 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.

1.30.15 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
type 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.

1.30.16 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.

1.30.17 Be careful with type equality

You generally want to use `isa()` and `<: (issubtype())` 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.

1.30.18 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)`.

1.31 Frequently Asked Questions

1.31.1 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 = 0`. The memory will be released the next time the garbage collector runs; you can force this to happen with `gc()`.

How can I modify the declaration of a type/immutable 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)
...
```

1.31.2 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
julia> function change_value!(y) # Create a new function
           y = 17
       end
julia> change_value!(x)
julia> x # x is unchanged!
10
```

In Julia, any function (including `change_value!()`) can’t change the binding of a local variable. If `x` (in the calling scope) is bound to an immutable object (like a real number), you can’t modify the object; likewise, if `x` is bound in the calling scope to a `Dict`, you can’t change it to be bound to an `ASCIIString`.

But here is a thing you should pay attention to: suppose `x` is bound to an `Array` (or any other mutable type). You cannot “unbind” `x` from this `Array`. But, since an `Array` is a *mutable* type, 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) # Create a new function
           A[1] = 5
       end
julia> change_array!(x)
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 Array. We passed `x` (which was previously bound to an Array) to the function. Notice that, after the function call, `x` is still bound to the same Array, but the content of that Array changed.

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.baz` 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...)
    @printf("%s\n", typeof(args))
    for (i, arg) in enumerate(args)
        @printf("Arg %d = %s\n", i, arg)
    end
end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
(Int64, Int64, Int64)
Arg 1 = 1
```

```
Arg 2 = 2
Arg 3 = 3
```

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as `<-...` instead of `...`.

... splits one argument into many different arguments in function calls

In contrast to the use of the `...` operator to denote slurping many different arguments into one argument when defining a function, the `...` operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of `...` is called *splatting*:

```
julia> function threeargs(a, b, c)
           @printf("a = %s::%s\n", a, typeof(a))
           @printf("b = %s::%s\n", b, typeof(b))
           @printf("c = %s::%s\n", c, typeof(c))
       end
threeargs (generic function with 1 method)

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

julia> threeargs(vec...)
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 `...`.

1.31.3 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:

```
function unstable(flag::Bool)
    if flag
        return 1
    else
        return 1.0
    end
end
```

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
  in sqrt at math.jl:128

julia> 2^-5
ERROR: DomainError
  in power_by_squaring at intfuncs.jl:70
  in ^ at intfuncs.jl:84
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `sqrt()`, most users want `sqrt(2.0)` to give a real number, and would be unhappy if it produced the complex number `1.4142135623730951 + 0.0im`. One could write the `sqrt()` function to switch to a complex-valued output only when passed a negative number (which is what `sqrt()` does in some other languages), but then the result would not be *type-stable* and the `sqrt()` function would have poor performance.

In these and other cases, you can get the result you want by choosing an *input type* that conveys your willingness to accept an *output type* in which the result can be represented:

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im

julia> 2.0^-5
0.03125
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> typemax{Int}
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `Int128` or `BigInt` in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, *type-stability is crucial* for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability is to merge the `Int` and `BigInt` types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem

under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach *can* be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

    9223372036854775807

>> int64(9223372036854775807) + 1

ans =

    9223372036854775807

>> int64(-9223372036854775808)

ans =

   -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

   -9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow *is* associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
julia> n = 2^62
4611686018427387904
```

```
julia> (n + 2n) >>> 1
6917529027641081856
```

See? No problem. That's the correct midpoint between 2^{62} and 2^{63} , despite the fact that `n + 2n` is - 4611686018427387904. Now try it in Matlab:

```
>> (n + 2*n)/2
```

```
ans =
```

```
4611686018427387904
```

Oops. Adding a `>>>` operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like $f(k) = 5k-1$. The machine code for this function is just this:

```
julia> code_native(f, (Int,))
.section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
    push    RBP
    mov RBP, RSP
Source line: 1
    lea RAX, QWORD PTR [RDI + 4*RDI - 1]
    pop RBP
    ret
```

The actual body of the function is a single `lea` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k,n)
    for i = 1:n
        k = f(k)
    end
    return k
end
g (generic function with 2 methods)

julia> code_native(g, (Int, Int))
.section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 3
    push    RBP
    mov RBP, RSP
    test    RSI, RSI
    jle 22
    mov EAX, 1
Source line: 3
    lea RDI, QWORD PTR [RDI + 4*RDI - 1]
```

```
    inc RAX
    cmp RAX, RSI
Source line: 2
    jle -17
Source line: 5
    mov RAX, RDI
    pop RBP
    ret
```

Since the call to `f` gets inlined, the loop body ends up being just a single `lea` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
julia> function g(k)
    for i = 1:10
        k = f(k)
    end
    return k
end
g (generic function with 2 methods)

julia> code_native(g, (Int,))
.section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 3
    push    RBP
    mov RBP, RSP
Source line: 3
    imul    RAX, RDI, 9765625
    add RAX, -2441406
Source line: 5
    pop RBP
    ret
```

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.

How do “abstract” or ambiguous fields in types interact with the compiler?

Types can be declared without specifying the types of their fields:

```
julia> type MyAmbiguousType
    a
end
```

This allows `a` to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType (constructor with 1 method)

julia> typeof(c)
MyAmbiguousType (constructor with 1 method)
```

`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> type MyType{T<:FloatingPoint}
    a::T
end
```

This is a better choice than

```
julia> type MyStillAmbiguousType
    a::FloatingPoint
end
```

because the first version specifies the type of `a` from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64} (constructor with 1 method)

julia> typeof(t)
MyStillAmbiguousType (constructor with 2 methods)
```

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 field `a`:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once `m` is constructed, the type of `m.a` cannot change:

```
julia> m.a = 4.5f0
4.5
```

```
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{FloatingPoint}(3.2)
MyType{FloatingPoint}(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, (MyType{Float64},))
code_llvm(func, (MyType{FloatingPoint},))
code_llvm(func, (MyType,))
```

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.

How should I declare “abstract container type” fields?

The same best practices that apply in the [previous section](#) also work for container types:

```
julia> type MySimpleContainer{A<:AbstractVector}
    a::A
end

julia> type MyAmbiguousContainer{T}
    a::AbstractVector{T}
end
```

For example:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}} (constructor with 1 method)

julia> c = MySimpleContainer([1:3]);

julia> typeof(c)
```

```
MySimpleContainer{Array{Int64,1}} (constructor with 1 method)
```

```
julia> b = MyAmbiguousContainer(1:3);
```

```
julia> typeof(b)
```

```
MyAmbiguousContainer{Int64} (constructor with 1 method)
```

```
julia> b = MyAmbiguousContainer([1:3]);
```

```
julia> typeof(b)
```

```
MyAmbiguousContainer{Int64} (constructor with 1 method)
```

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:

```
function sumfoo(c::MySimpleContainer)
    s = 0
    for x in c.a
        s += foo(x)
    end
    s
end
```

```
foo(x::Integer) = x
```

```
foo(x::FloatingPoint) = round(x)
```

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 of `a`. You could do it like this:

```
function myfun{T<:FloatingPoint}(c::MySimpleContainer{Vector{T}})
    ...
end
function myfun{T<:Integer}(c::MySimpleContainer{Vector{T}})
    ...
end
```

This works fine for `Vector{T}`, but we'd also have to write explicit versions for `UnitRange{T}` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `MyContainer`:

```
type MyContainer{T, A<:AbstractVector}
```

```
    a::A
```

```
end
```

```
MyContainer(v::AbstractVector) = MyContainer{eltype(v), typeof(v)}(v)
```

```
julia> b = MyContainer(1.3:5);
```

```
julia> typeof(b)
```

```
MyContainer{Float64, UnitRange{Float64}}
```

Note the somewhat surprising fact that `T` doesn't appear in the declaration of field `a`, a point that we'll return to in a moment. With this approach, one can write functions such as:

```
function myfunc{T<:Integer, A<:AbstractArray}(c::MyContainer{T,A})
    return c.a[1]+1
end
# Note: because we can only define MyContainer for
# A<:AbstractArray, and any unspecified parameters are arbitrary,
# the previous could have been written more succinctly as
#     function myfunc{T<:Integer}(c::MyContainer{T})

function myfunc{T<:FloatingPoint}(c::MyContainer{T})
    return c.a[1]+2
end

function myfunc{T<:Integer}(c::MyContainer{T,Vector{T}})
    return c.a[1]+3
end

julia> myfunc(MyContainer{1:3})
2

julia> myfunc(MyContainer{1.0:3})
3.0

julia> myfunc(MyContainer{[1:3]})
4
```

As you can see, with this approach it's possible to specialize on both the element type `T` and the array type `A`.

However, there's one remaining hole: we haven't enforced that `A` has element type `T`, so it's perfectly possible to construct an object like this:

```
julia> b = MyContainer{Int64, UnitRange{Float64}}(1.3:5);

julia> typeof(b)
MyContainer{Int64,UnitRange{Float64}}
```

To prevent this, we can add an inner constructor:

```
type MyBetterContainer{T<:Real, A<:AbstractVector}
    a::A

    MyBetterContainer(v::AbstractVector{T}) = new(v)
end
MyBetterContainer(v::AbstractVector) = MyBetterContainer{eltype(v),typeof(v)}(v)

julia> b = MyBetterContainer(1.3:5);

julia> typeof(b)
MyBetterContainer{Float64,UnitRange{Float64}}

julia> b = MyBetterContainer{Int64, UnitRange{Float64}}(1.3:5);
ERROR: no method MyBetterContainer{UnitRange{Float64},}
```

The inner constructor requires that the element type of `A` be `T`.

1.31.4 Nothingness and missing values

How does “null” or “nothingness” work in Julia?

Unlike many languages (for example, C and Java), Julia does not have a “null” value. 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` function.

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`.

Note that `Nothing` (uppercase) is the type of `nothing`, and should only be used in a context where a type is required (e.g. a declaration).

You may occasionally see `None`, which is quite different. It is the empty (or “bottom”) type, a type with no values and no subtypes (except itself). You will generally not need to use this type.

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.

1.31.5 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.

1.31.6 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 <http://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.

1.32 Noteworthy Differences from other Languages

1.32.1 Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is in no way 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:

- Arrays are indexed with square brackets, `A[i, j]`.
- Arrays are assigned by reference. After `A=B`, assigning into `B` will modify `A` as well.
- Values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller.
- Matlab combines allocation and assignment into single statements, e.g., `a(4) = 3.2` creates the array `a = [0 0 0 3.2]` and `a(5) = 7` grows it. Julia separates allocation and assignment: if `a` is of length 4, `a[5] = 7` yields an error. Julia has a `push!` function which grows `Vectors` much more efficiently than Matlab's `a(end+1) = val`.
- The imaginary unit `sqrt(-1)` is represented in julia with `im`.
- Literal numbers without a decimal point (such as `42`) create integers instead of floating point numbers. Arbitrarily large integer literals are supported. But this means that 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).

- **Multiple values are returned and assigned with parentheses, e.g.** `return (a, b)` and `(a, b) = f(x)`. The equivalent of `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 1-dimensional arrays. Column vectors are of size N , not $N \times 1$. For example, `rand(N)` makes a 1-dimensional array.
- Concatenating scalars and arrays with the syntax `[x, y, z]` concatenates in the first dimension (“vertically”). For the second dimension (“horizontally”), use spaces as in `[x y z]`. To construct block matrices (concatenating in the first two dimensions), the syntax `[a b; c d]` is used to avoid confusion.
- Colons `a:b` and `a:b:c` construct `Range` objects. To construct a full vector, use `linspace`, or “concatenate” the range by enclosing it in brackets, `[a:b]`.
- Functions return values using the `return` keyword, instead of by listing their names in the function definition (see [The return Keyword](#) for details).
- A file may contain any number of functions, and all definitions will be externally visible when the file is loaded.
- 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)`.
- Functions such as `sort` that operate column-wise by default (`sort(A)` is equivalent to `sort(A, 1)`) do not have special behavior for $1 \times N$ arrays; the argument is returned unmodified since it still performs `sort(A, 1)`. To sort a $1 \times N$ matrix like a vector, use `sort(A, 2)`.
- If `A` is a 2-dimensional array `fft(A)` computes a 2D FFT. In particular, it is not equivalent to `fft(A, 1)`, which computes a 1D FFT acting column-wise.
- Parentheses must be used to call a function with zero arguments, as in `tic()` and `toc()`.
- Do not use 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. The function `println` can be used to print a value followed by a newline.
- If `A` and `B` are arrays, `A == B` doesn’t return an array of booleans. Use `A .== B` instead. Likewise for the other boolean operators, `<`, `>`, `!=`, etc.
- The operators `&`, `|`, and `$` perform the bitwise operations `and`, `or`, and `xor`, respectively, and have precedence similar to Python’s bitwise operators (not like C). They can operate on scalars or elementwise 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)`).
- The elements of a collection can be passed as arguments to a function using `...`, as in `xs=[1,2]; f(xs...)`.
- Julia’s `svd` returns singular values as a vector instead of as a full diagonal matrix.
- In Julia, `...` is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.
- The variable `ans` is set to the value of the last expression issued in an interactive session, but not set when Julia code is run in other ways.
- The closest analog to Julia’s `types` are Matlab’s `classes`. Matlab’s `structs` behave somewhere between Julia’s `types` and `Dicts`; in particular, if you need to be able to add fields to a `struct` on-the-fly, use a `Dict` rather than a `type`.

1.32.2 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 uses `=` for assignment. Julia does not provide any operator like `<-` or `<<-`.
- Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.
- Julia's matrix operations are more like traditional mathematical notation than R's. If `A` and `B` are matrices, then `A * B` defines a matrix multiplication in Julia equivalent to R's `A %*% B`. In R, this same notation would perform an elementwise Hadamard product. To get the elementwise multiplication operation, you need to write `A .* B` in Julia.
- Julia performs matrix transposition using the `'` operator. Julia's `A'` is therefore equivalent to R's `t(A)`.
- Julia does not require parentheses when writing `if` statements or `for` 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`.
- Julia does not provide `nrow` and `ncol`. Instead, use `size(M, 1)` for `nrow(M)` and `size(M, 2)` for `ncol(M)`.
- Julia's SVD is not thinned by default, unlike R. To get results like R's, you will often want to call `svd(X, true)` on a matrix `X`.
- Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `c(1)` are the same. In Julia, they can not be used interchangeably. One potentially confusing result of this is that `x' * y` for vectors `x` and `y` is a 1-element vector, not a scalar. To get a scalar, use `dot(x, y)`.
- Julia's `diag()` and `diagm()` are not like R's.
- Julia cannot assign to the results of function calls on the left-hand of an assignment operation: you cannot write `diag(M) = ones(n)`.
- Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in [packages](#) like the [DataFrames](#) and [Distributions](#) packages:
 - Distributions functions are found in 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's lists. When returning multiple items, you should typically use a tuple: instead of `list(a = 1, b = 2)`, use `(1, 2)`.
- Julia encourages all users to write their own types. Julia's types are much 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 passed and assigned by reference. 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.
- Concatenation of vectors and matrices is done using `hcat` and `vcat`, not `c`, `rbind` and `cbind`.
- A Julia range object like `a:b` is not shorthand for a vector like in R, but is a specialized type of object that is used for iteration without high memory overhead. To convert a range into a vector, you need to wrap the range with brackets `[a:b]`.

- `max`, `min` are the equivalent of `pmax` and `pmin` in R, but both arguments need to have the same dimensions. While `maximum`, `minimum` replace `max` and `min` in R, there are important differences.
- The functions `sum`, `prod`, `maximum`, `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, 1)` is a row vector containing the sum over each column and `sum(A, 2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where `sum(B, 1)=11` and `sum(B, 2)=12`. If the second argument is a vector, then it specifies all the dimensions over which the sum is performed, e.g., `sum(A, [1, 2])=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 `sort(v)` and `sort!(v)`.
- `colMeans()` and `rowMeans()`, `size(m, 1)` and `size(m, 2)`
- In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.
- Unlike R, there is no delayed evaluation in Julia. For most users, this means that there are very few unquoted expressions or column names.
- Julia does not support the `NULL` type.
- There is no equivalent of R's `assign` or `get` in Julia.

1.32.3 Noteworthy differences from Python

- Indexing of arrays, strings, etc. in Julia is 1-based not 0-based.
- The last element of a list or array is indexed with `end` in Julia, not `-1` as in Python.
- Comprehensions in Julia do not (yet) have the optional `if` clause found in Python.
- For, `if`, `while`, etc. blocks in Julia are terminated by `end`; indentation is not significant.
- 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 evaluates default values of function arguments every time the method is invoked (not once when the function is defined as in Python). This means that function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand function `g(x=[1, 2]) = push!(x, 3)` returns `[1, 2, 3]` every time it is called as `g()`.

1.33 Unicode Input

Please see the online documentation.

The Julia Standard Library

2.1 Essentials

2.1.1 Introduction

The Julia standard library 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:

- Except for functions in built-in modules (`Pkg`, `Collections`, `Graphics`, `Test` and `Profile`), all functions documented here are directly available for use in programs.
- 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.

2.1.2 Getting Around

exit (*[code]*)

Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

quit ()

Quit the program indicating that the processes completed successfully. This function calls `exit(0)` (see `exit()`).

atexit (*f*)

Register a zero-argument function to be called at exit.

isinteractive () → Bool

Determine whether Julia is running an interactive session.

whos (*[Module,] [pattern::Regex]*)

Print information about global variables in a module, optionally restricted to those matching *pattern*.

edit (*file::String*, *[line]*)

Edit a file optionally providing a line number to edit at. Returns to the julia prompt when you quit the editor.

edit (*function* [, *types*])

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit.

@edit ()

Evaluates the arguments to the function call, determines their types, and calls the `edit` function on the resulting expression

less (*file::String* [, *line*])

Show a file using the default pager, optionally providing a starting line number. Returns to the julia prompt when you quit the pager.

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.

@less ()

Evaluates the arguments to the function call, determines their types, and calls the `less` function on the resulting expression

clipboard (*x*)

Send a printed form of *x* to the operating system clipboard (“copy”).

clipboard () → *String*

Return a string with the contents of the operating system clipboard (“paste”).

require (*file::String...*)

Load source files once, 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 in the current working directory, then looks for package code under `Pkg.dir()`, then tries paths in the global array `LOAD_PATH`.

reload (*file::String*)

Like `require`, except forces loading of files regardless of whether they have been loaded before. Typically used when interactively developing libraries.

include (*path::String*)

Evaluate the contents of a source file in the current context. 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. All paths refer to files on node 1 when running in parallel, and files will be fetched from node 1. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

include_string (*code::String*)

Like `include`, except reads code from the given string rather than from a file. Since there is no file path involved, no path processing or fetching from node 1 is done.

help (*name*)

Get help for a function. *name* can be an object or a string.

apropos (*string*)

Search documentation for functions related to *string*.

which (*f*, *types*)

Return the method of *f* (a `Method` object) that will be called for arguments with the given types.

@which ()

Evaluates the arguments to the function call, determines their types, and calls the `which` function on the resulting expression

methods (*f* [, *types*])

Show all methods of *f* with their argument types.

If *types* is specified, an array of methods whose types match is returned.

methodswith (*typ* [, *showparents*])

Return an array of methods with an argument of type *typ*. If optional *showparents* is *true*, also return arguments with a parent type of *typ*, excluding type *Any*.

@show ()

Show an expression and result, returning the result

versioninfo ([*verbose::Bool*])

Print information about the version of Julia in use. If the *verbose* argument is *true*, detailed system information is shown as well.

workspace ()

Replace the top-level module (*Main*) with a new one, providing a clean workspace. The previous *Main* module is made available as *LastMain*. A previously-loaded package can be accessed using a statement such as `using LastMain.Package`.

This function should only be used interactively.

2.1.3 All Objects

is (*x*, *y*) → *Bool*

=== (*x*, *y*) → *Bool*

≡ (*x*, *y*) → *Bool*

Determine whether *x* and *y* are identical, in the sense that no program could distinguish them. Compares mutable objects by address in memory, and compares immutable objects (such as numbers) by contents at the bit level. This function is sometimes called *egal*.

isa (*x*, *type*) → *Bool*

Determine whether *x* is of the given *type*.

isequal (*x*, *y*)

Similar to `==`, except treats all floating-point NaN values as equal to each other, and treats `-0.0` as unequal to `0.0`. For values that are not floating-point, `isequal` calls `==` (so that if you define a `==` method for a new type you automatically get `isequal`).

`isequal` is the comparison function used by hash tables (*Dict*). `isequal(x, y)` must imply that `hash(x) == hash(y)`.

This typically means that if you define your own `==` function then you must define a corresponding `hash` (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 `==`).

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. This is the default comparison used by `sort`. 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.

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.

lexcmp (*x*, *y*)

Compare *x* and *y* lexicographically and return -1, 0, or 1 depending on whether *x* is less than, equal to, or greater than *y*, respectively. This function should be defined for lexicographically comparable types, and `lexless` will call `lexcmp` by default.

lexless (*x*, *y*)

Determine whether *x* is lexicographically less than *y*.

typeof (*x*)

Get the concrete type of *x*.

tuple (*xs...*)

Construct a tuple of the given objects.

ntuple (*n*, *f::Function*)

Create a tuple of length *n*, computing each element as `f(i)`, where *i* is the index of the element.

object_id (*x*)

Get a unique integer id for *x*. `object_id(x) == object_id(y)` if and only if `is(x, y)`.

hash (*x* [, *h*])

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.

finalizer (*x*, *function*)

Register a function `f(x)` to be called when there are no program-accessible references to *x*. The behavior of this function is unpredictable if *x* is of a bits type.

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.

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.

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::ObjectIdDict)` (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.

isdefined ([*object*], *index* | *symbol*)

Tests whether an assignable location is defined. The arguments can be an array and index, a composite object and field name (as a symbol), or a module and a symbol. With a single symbol argument, tests whether a global variable with that name is defined in `current_module()`.

convert (*type*, *x*)

Try to convert *x* to the given type. Conversions from floating point to integer, rational to integer, and complex to real will raise an `InexactError` if *x* cannot be represented exactly in the new type.

promote (*xs...*)

Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

oftype (*x*, *y*)Convert *y* to the type of *x*.**widen** (*type* | *x*)

If the argument is a type, return a “larger” type (for numeric types, this will be a type with at least as much range and precision as the argument, and usually more). Otherwise the argument *x* is converted to `widen(typeof(x))`.

```
julia> widen{Int32}
Int64
```

```
julia> widen{1.5f0}
1.5
```

identity (*x*)

The identity function. Returns its argument.

2.1.4 Types

super (*T::DataType*)Return the supertype of *DataType* *T***issubtype** (*type1*, *type2*)

True if and only if all values of *type1* are also of *type2*. Can also be written using the `<:` infix operator as *type1* `<:` *type2*.

`<:` (*T1*, *T2*)Subtype operator, equivalent to `issubtype(T1, T2)`.**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.

subtypetree (*T::DataType*)

Return a nested list of all subtypes of *DataType* *T*. Note that all currently loaded subtypes are included, including those not visible in the current module.

typemin (*type*)

The lowest value representable by the given (real) numeric type.

typemax (*type*)

The highest value representable by the given (real) numeric type.

realmin (*type*)

The smallest in absolute value non-subnormal value representable by the given floating-point type

realmax (*type*)

The highest finite value representable by the given floating-point type

maxintfloat (*type*)

The largest integer losslessly representable by the given floating-point type

sizeof (*type*)

Size, in bytes, of the canonical binary representation of the given type, if any.

eps ([*type*])

The distance between 1.0 and the next larger representable floating-point value of *type*. Only floating-point types are sensible arguments. If *type* is omitted, then `eps{Float64}` is returned.

eps (*x*)The distance between *x* and the next larger representable floating-point value of the same type as *x*.

promote_type(*type1*, *type2*)

Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists to which both types can be promoted losslessly, 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.

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.

getfield(*value*, *name::Symbol*)

Extract a named field from a value of composite type. The syntax `a.b` calls `getfield(a, :b)`, and the syntax `a.(b)` calls `getfield(a, b)`.

setfield!(*value*, *name::Symbol*, *x*)

Assign `x` to a named field in `value` of composite type. The syntax `a.b = c` calls `setfield!(a, :b, c)`, and the syntax `a.(b) = c` calls `setfield!(a, b, c)`.

fieldoffsets(*type*)

The byte offset of each field of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct type:

```
julia> structinfo(T) = [zip(fieldoffsets(T),names(T),T.types)...];

julia> structinfo(StatStruct)
12-element Array{(Int64,Symbol,DataType),1}:
 (0, :device, UInt64)
 (8, :inode, UInt64)
 (16, :mode, UInt64)
 (24, :nlink, Int64)
 (32, :uid, UInt64)
 (40, :gid, UInt64)
 (48, :rdev, UInt64)
 (56, :size, Int64)
 (64, :blksize, Int64)
 (72, :blocks, Int64)
 (80, :mtime, Float64)
 (88, :ctime, Float64)
```

fieldtype(*value*, *name::Symbol*)

Determine the declared type of a named field in a value of composite type.

isimmutable(*v*)

True if value `v` is immutable. See *Immutable 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.

isbits(*T*)

True if `T` is a “plain data” type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as `UInt8`, `Float64`, and `Complex{Float64}`.

```
julia> isbits(Complex{Float64})
true

julia> isbits(Complex)
false
```

isleaftype(*T*)

Determine whether `T` is a concrete type that can have instances, meaning its only subtypes are itself and `None` (but `T` itself is not `None`).

typejoin (*T*, *S*)Compute a type that contains both *T* and *S*.**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.

2.1.5 Generic Functions

apply (*f*, *x*...)Accepts a function and several arguments, each of which must be iterable. The elements generated by all the arguments are appended into a single list, which is then passed to *f* as its argument list.

```
julia> function f(x, y) # Define a function f
           x + y
       end;
```

```
julia> apply(f, [1 2]) # Apply f with 1 and 2 as arguments
3
```

`apply` is called to implement the ... argument splicing syntax, and is usually not called directly:
`apply(f, x) === f(x...)`

method_exists (*f*, *tuple*) → Bool

Determine whether the given generic function has a method matching the given tuple of argument types.

```
julia> method_exists(length, (Array,))
true
```

applicable (*f*, *args*...) → Bool

Determine whether the given generic function has a method applicable to the given arguments.

```
julia> function f(x, y)
           x + y
       end;
```

```
julia> applicable(f, 1)
false
```

```
julia> applicable(f, 1, 2)
true
```

invoke (*f*, (*types*...), *args*...)

Invoke a method for the given generic function matching the specified types (as a tuple), on the specified arguments. The arguments must be compatible with the specified types. This 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).

|> (*x*, *f*)

Applies a function to the preceding argument. This allows for easy function chaining.

```
julia> [1:5] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

2.1.6 Syntax

eval (*[m::Module]*, *expr::Expr*)

Evaluate an expression in the given module and return the result. Every module (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

@eval ()

Evaluate an expression and return the value.

evalfile (*path::String*)

Evaluate all expressions in the given file, and return the value of the last one. No other processing (path searching, fetching from node 1, etc.) is performed.

esc (*e::ANY*)

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 *Non-Standard String Literals* section of the Metaprogramming chapter of the manual for more details and examples.

gensym (*[tag]*)

Generates a symbol which will not conflict with other variable names.

@gensym ()

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

parse (*str, start; greedy=true, raise=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.

parse (*str; raise=true*)

Parse the whole 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.

2.1.7 System

run (*command*)

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

spawn (*command*)

Run a command object asynchronously, returning the resulting `Process` object.

DevNull

Used in a stream redirect to discard all data written to it. Essentially equivalent to `/dev/null` on Unix or `NUL` on Windows. Usage: `run(`cat test.txt` |> DevNull)`

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.

process_running (*p::Process*)

Determine whether a process is currently running.

process_exited (*p::Process*)

Determine whether a process has exited.

kill (*p::Process, signal=SIGTERM*)

Send a signal to a process. The default is to terminate the process.

open (*command, mode::String="r", stdio=DevNull*)

Start running *command* asynchronously, and return a tuple (*stream, process*). If *mode* is "r", then *stream* reads from the process's standard output and *stdio* optionally specifies the process's standard input stream. If *mode* is "w", then *stream* writes to the process's standard input and *stdio* optionally specifies the process's standard output stream.

open (*f::Function, command, mode::String="r", stdio=DevNull*)

Similar to `open(command, mode, stdio)`, but calls *f(stream)* on the resulting read or write stream, then closes the stream and waits for the process to complete. Returns the value returned by *f*.

readandwrite (*command*)

Starts running a command asynchronously, and returns a tuple (*stdout, stdin, process*) of the output stream and input stream of the process, and the process object itself.

ignorestatus (*command*)

Mark a command object so that running it will not throw an error if the result code is non-zero.

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.

setenv (*command, env; dir=working_dir*)

Set environment variables to use when running the given command. *env* is either a dictionary mapping strings to strings, or an array of strings of the form "*var=val*".

The *dir* keyword argument can be used to specify a working directory for the command.

|> (*command, command*)

|> (*command, filename*)

|> (*filename, command*)

Redirect operator. Used for piping the output of a process into another (first form) or to redirect the standard output/input of a command to/from a file (second and third forms).

Examples:

- `run(`ls` |> `grep xyz`)`
- `run(`ls` |> "out.txt")`
- `run("out.txt" |> `grep xyz`)`

>> (*command, filename*)

Redirect standard output of a process, appending to the destination file.

.> (*command, filename*)

Redirect the standard error stream of a process.

gethostname () → String

Get the local machine's host name.

getipaddr () → String

Get the IP address of the local machine, as a string of the form "x.x.x.x".

getpid () → Int32

Get julia's process ID.

time (*[t::TmStruct]*)

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution. When passed a `TmStruct`, converts it to a number of seconds since the epoch.

time_ns ()

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

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.

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.

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`.

tic ()

Set a timer to be read by the next call to `toc` () or `toq` (). The macro call `@time expr` can also be used to time evaluation.

toc ()

Print and return the time elapsed since the last `tic` () .

toq ()

Return, but do not print, the time elapsed since the last `tic` () .

@time ()

A macro to execute an expression, printing the time it took to execute and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

@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.

@allocated ()

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

EnvHash () → EnvHash

A singleton of this type provides a hash table interface to environment variables.

ENV

Reference to the singleton `EnvHash`, providing a dictionary interface to system environment variables.

@unix ()

Given `@unix? a : b`, do `a` on Unix systems (including Linux and OS X) and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@osx ()

Given `@osx? a : b`, do `a` on OS X and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@linux ()

Given `@linux? a : b`, do `a` on Linux and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@windows()

Given `@windows? a : b`, do `a` on Windows and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

2.1.8 Errors**error** (*message::String*)

Raise an error with the given message

throw (*e*)

Throw an object as an exception

rethrow (*[e]*)

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a `catch` block).

backtrace ()

Get a backtrace object for the current program point.

catch_backtrace ()

Get the backtrace of the current exception, for use within `catch` blocks.

assert (*cond* [, *text*])

Raise an error if `cond` is false. Also available as the macro `@assert expr`.

@assert ()

Raise an error if `cond` is false. Preferred syntax for writings assertions.

ArgumentError

The parameters given to a function call are not valid.

BoundsError

An indexing operation into an array tried to access an out-of-bounds element.

EOFError

No more data was available to read from a file or stream.

ErrorException

Generic error type. The error message, in the `.msg` field, may provide more specific details.

KeyError

An indexing operation into an Associative (`Dict`) or `Set` like object tried to access or delete a non-existent element.

LoadError

An error occurred while *including*, *requiring*, or *using* a file. The error specifics should be available in the `.error` field.

MethodError

A method with the required type signature does not exist in the given generic function.

ParseError

The expression passed to the `parse` function could not be interpreted as a valid Julia expression.

ProcessExitedException

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

SystemError

A system call failed with an error code (in the `errno` global variable).

TypeError

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

2.1.9 Events

Timer (*f::Function*)

Create a timer to call the given callback function. The callback is passed one argument, the timer object itself. The timer can be started and stopped with `start_timer` and `stop_timer`.

start_timer (*t::Timer*, *delay*, *repeat*)

Start invoking the callback for a `Timer` after the specified initial delay, and then repeating with the given interval. Times are in seconds. If `repeat` is 0, the timer is only triggered once.

stop_timer (*t::Timer*)

Stop invoking the callback for a timer.

2.1.10 Reflection

module_name (*m::Module*) → Symbol

Get the name of a module as a symbol.

module_parent (*m::Module*) → Module

Get a module's enclosing module. `Main` is its own parent.

current_module () → Module

Get the *dynamically* current module, which is the module code is currently being read from. In general, this is not the same as the module containing the call to this function.

fullname (*m::Module*)

Get the fully-qualified name of a module as a tuple of symbols. For example, `fullname(Base.Pkg)` gives `(:Base, :Pkg)`, and `fullname(Main)` gives `()`.

names (*x::Module* [, *all=false* [, *imported=false*]])

Get an array of the names exported by a module, with optionally more module globals according to the additional parameters.

names (*x::DataType*)

Get an array of the fields of a data type.

isconst ([*m::Module*], *s::Symbol*) → Bool

Determine whether a global is declared `const` in a given module. The default module argument is `current_module()`.

isgeneric (*f::Function*) → Bool

Determine whether a function is generic.

function_name (*f::Function*) → Symbol

Get the name of a generic function as a symbol, or `:anonymous`.

function_module (*f::Function*, *types*) → Module

Determine the module containing a given definition of a generic function.

functionloc (*f::Function*, *types*)

Returns a tuple (`filename`, `line`) giving the location of a method definition.

functionlocs (*f::Function*, *types*)

Returns an array of the results of `functionloc` for all matching definitions.

2.1.11 Internals

gc ()

Perform garbage collection. This should not generally be used.

gc_disable()

Disable garbage collection. This should be used only with extreme caution, as it can cause memory use to grow without bound.

gc_enable()

Re-enable garbage collection after calling `gc_disable`.

macroexpand(x)

Takes the expression `x` and returns an equivalent expression with all macros removed (expanded).

expand(x)

Takes the expression `x` and returns an equivalent expression in lowered form

code_lowered(f, types)

Returns an array of lowered ASTs for the methods matching the given generic function and type signature.

@code_lowered()

Evaluates the arguments to the function call, determines their types, and calls the `code_lowered` function on the resulting expression

code_typed(f, types)

Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature.

@code_typed()

Evaluates the arguments to the function call, determines their types, and calls the `code_typed` function on the resulting expression

code_llvm(f, types)

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to STDOUT.

@code_llvm()

Evaluates the arguments to the function call, determines their types, and calls the `code_llvm` function on the resulting expression

code_native(f, types)

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to STDOUT.

@code_native()

Evaluates the arguments to the function call, determines their types, and calls the `code_native` function on the resulting expression

precompile(f, args::(Any...,))

Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

2.2 Collections and Data Structures

2.2.1 Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`. The general `for` loop:

```
for i = I      # or "for i in I"
    # body
end
```

is translated to:

```
state = start(I)
while !done(I, state)
    (i, state) = next(I, state)
    # body
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type.

start (*iter*) → state
Get initial iteration state for an iterable object

done (*iter, state*) → Bool
Test whether we are done iterating

next (*iter, state*) → item, state
For a given iterable object and iteration state, return the current item and the next iteration state

zip (*iters...*)
For a set of iterable objects, returns an iterable of tuples, where the *i*th tuple contains the *i*th component of each input iterable.

Note that `zip` is its own inverse: `[zip(zip(a...)...)...] == [a...]`.

enumerate (*iter*)
Return an iterator that yields (*i*, *x*) where *i* is an index starting at 1, and *x* is the *i*th value from the given iterator. It's useful when you need not only the values *x* over which you are iterating, but also the index *i* of the iterations.

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
    println("$index $value")
end

1 a
2 b
3 c
```

Fully implemented by: `Range`, `UnitRange`, `NDRange`, `Tuple`, `Real`, `AbstractArray`, `IntSet`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `EachLine`, `String`, `Set`, `Task`.

2.2.2 General Collections

isempty (*collection*) → Bool
Determine whether a collection is empty (has no elements).

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

empty! (*collection*) → collection
Remove all elements from a collection.

length (*collection*) → Integer
For ordered, indexable collections, the maximum index *i* for which `getindex(collection, i)` is valid.
For unordered collections, the number of elements.

endof (*collection*) → Integer
Returns the last index of the collection.

```
julia> endof([1,2,4])
3
```

Fully implemented by: `Range`, `UnitRange`, `Tuple`, `Number`, `AbstractArray`, `IntSet`, `Dict`, `WeakKeyDict`, `String`, `Set`.

2.2.3 Iterable Collections

in (*item*, *collection*) → Bool

\in (*item*, *collection*) → Bool

\ni (*collection*, *item*) → Bool

\notin (*item*, *collection*) → Bool

\ni (*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. Some collections need a slightly different definition; for example `Sets` check whether the item is `isequal` to one of the elements. Dicts look for (*key*, *value*) pairs, and the key is compared using `isequal`. To test for the presence of a key in a dictionary, use `haskey` or `k in keys(dict)`.

eltype (*collection*)

Determine the type of the elements generated by iterating *collection*. For associative collections, this will be a (*key*, *value*) tuple type.

indexin (*a*, *b*)

Returns a vector containing the highest index in *b* for each value in *a* that is a member of *b*. The output vector contains 0 wherever *a* is not a member of *b*.

findin (*a*, *b*)

Returns the indices of elements in collection *a* that appear in collection *b*

unique (*itr* [, *dim*])

Returns an array containing only the unique elements of the iterable *itr*, in the order that the first of each set of equivalent elements originally appears. If *dim* is specified, returns unique regions of the array *itr* along *dim*.

reduce (*op*, *v0*, *itr*)

Reduce the given collection *itr* with the given binary operator *op*. *v0* must be a neutral element for *op* that will be returned for empty collections. It is unspecified whether *v0* is used for non-empty collections.

Reductions for certain commonly-used operators have special implementations which 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, and parallelism will also 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.

reduce (*op*, *itr*)

Like `reduce(op, v0, itr)`. This cannot be used with empty collections, except for some special cases (e.g. when *op* is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of *op*.

foldl (*op*, *v0*, *itr*)

Like `reduce`, but with guaranteed left associativity. *v0* will be used exactly once.

foldl (*op*, *itr*)

Like `foldl(op, v0, itr)`, but using the first element of *itr* as *v0*. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

foldr (*op*, *v0*, *itr*)

Like `reduce`, but with guaranteed right associativity. *v0* will be used exactly once.

foldr (*op*, *itr*)

Like `foldr` (*op*, *v0*, *itr*), but using the last element of *itr* as *v0*. In general, this cannot be used with empty collections (see `reduce` (*op*, *itr*)).

maximum (*itr*)

Returns the largest element in a collection.

maximum (*A*, *dims*)

Compute the maximum value of an array over the given dimensions.

maximum! (*r*, *A*)

Compute the maximum value of *A* over the singleton dimensions of *r*, and write results to *r*.

minimum (*itr*)

Returns the smallest element in a collection.

minimum (*A*, *dims*)

Compute the minimum value of an array over the given dimensions.

minimum! (*r*, *A*)

Compute the minimum value of *A* over the singleton dimensions of *r*, and write results to *r*.

extrema (*itr*)

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

indmax (*itr*) → Integer

Returns the index of the maximum element in a collection.

indmin (*itr*) → Integer

Returns the index of the minimum element in a collection.

findmax (*itr*) → (*x*, *index*)

Returns the maximum element and its index.

findmax (*A*, *dims*) → (*maxval*, *index*)

For an array input, returns the value and index of the maximum over the given dimensions.

findmin (*itr*) → (*x*, *index*)

Returns the minimum element and its index.

findmin (*A*, *dims*) → (*minval*, *index*)

For an array input, returns the value and index of the minimum over the given dimensions.

maxabs (*itr*)

Compute the maximum absolute value of a collection of values.

maxabs (*A*, *dims*)

Compute the maximum absolute values over given dimensions.

maxabs! (*r*, *A*)

Compute the maximum absolute values over the singleton dimensions of *r*, and write values to *r*.

minabs (*itr*)

Compute the minimum absolute value of a collection of values.

minabs (*A*, *dims*)

Compute the minimum absolute values over given dimensions.

minabs! (*r*, *A*)

Compute the minimum absolute values over the singleton dimensions of *r*, and write values to *r*.

sum (*itr*)
Returns the sum of all elements in a collection.

sum (*A*, *dims*)
Sum elements of an array over the given dimensions.

sum! (*r*, *A*)
Sum elements of *A* over the singleton dimensions of *r*, and write results to *r*.

sum (*f*, *itr*)
Sum the results of calling function *f* on each element of *itr*.

sumabs (*itr*)
Sum absolute values of all elements in a collection. This is equivalent to *sum(abs(itr))* but faster.

sumabs (*A*, *dims*)
Sum absolute values of elements of an array over the given dimensions.

sumabs! (*r*, *A*)
Sum absolute values of elements of *A* over the singleton dimensions of *r*, and write results to *r*.

sumabs2 (*itr*)
Sum squared absolute values of all elements in a collection. This is equivalent to *sum(abs2(itr))* but faster.

sumabs2 (*A*, *dims*)
Sum squared absolute values of elements of an array over the given dimensions.

sumabs2! (*r*, *A*)
Sum squared absolute values of elements of *A* over the singleton dimensions of *r*, and write results to *r*.

prod (*itr*)
Returns the product of all elements of a collection.

prod (*A*, *dims*)
Multiply elements of an array over the given dimensions.

prod! (*r*, *A*)
Multiply elements of *A* over the singleton dimensions of *r*, and write results to *r*.

any (*itr*) → Bool
Test whether any elements of a boolean collection are true.

any (*A*, *dims*)
Test whether any values along the given dimensions of an array are true.

any! (*r*, *A*)
Test whether any values in *A* along the singleton dimensions of *r* are true, and write results to *r*.

all (*itr*) → Bool
Test whether all elements of a boolean collection are true.

all (*A*, *dims*)
Test whether all values along the given dimensions of an array are true.

all! (*r*, *A*)
Test whether all values in *A* along the singleton dimensions of *r* are true, and write results to *r*.

count (*p*, *itr*) → Integer
Count the number of elements in *itr* for which predicate *p* returns true.

any (*p*, *itr*) → Bool
Determine whether predicate *p* returns true for any elements of *itr*.

all (*p*, *itr*) → Bool

Determine whether predicate *p* returns true for all elements of *itr*.

```
julia> all(i->(4<=i<=6), [4,5,6])
true
```

map (*f*, *c...*) → collection

Transform collection *c* by applying *f* to each element. For multiple collection arguments, apply *f* elementwise.

```
julia> map((x) -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
11
22
33
```

map! (*function*, *collection*)

In-place version of `map()`.

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.

mapreduce (*f*, *op*, *v0*, *itr*)

Apply function *f* to each element in *itr*, and then reduce the result using the binary function *op*. *v0* must be a neutral element for *op* that will be returned for empty collections. It is unspecified whether *v0* is used for non-empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, v0, map(f, itr))`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

```
julia> mapreduce(x->x^2, +, [1:3]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Use `mapfoldl()` or `mapfoldr()` instead for guaranteed left or right associativity.

mapreduce (*f*, *op*, *itr*)

Like `mapreduce(f, op, v0, itr)`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

mapfoldl (*f*, *op*, *v0*, *itr*)

Like `mapreduce`, but with guaranteed left associativity. *v0* will be used exactly once.

mapfoldl (*f*, *op*, *itr*)

Like `mapfoldl(f, op, v0, itr)`, but using the first element of *itr* as *v0*. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

mapfoldr (*f*, *op*, *v0*, *itr*)

Like `mapreduce`, but with guaranteed right associativity. *v0* will be used exactly once.

mapfoldr (*f*, *op*, *itr*)

Like `mapfoldr(f, op, v0, itr)`, but using the first element of *itr* as *v0*. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

first (*coll*)

Get the first element of an iterable collection.

last (*coll*)

Get the last element of an ordered collection, if it can be computed in $O(1)$ time. This is accomplished by calling `endof` to get the last index.

step (*r*)

Get the step size of a `Range` object.

collect (*collection*)

Return an array of all items in a collection. For associative collections, returns (key, value) tuples.

collect (*element_type*, *collection*)

Return an array of type `Array{element_type, 1}` of all items in a collection.

issubset (*a*, *b*) $\subseteq (A, S) \rightarrow \text{Bool}$ $\not\subseteq (A, S) \rightarrow \text{Bool}$ $\subsetneq (A, S) \rightarrow \text{Bool}$

Determine whether every element of *a* is also in *b*, using the `in` function.

filter (*function*, *collection*)

Return a copy of *collection*, removing elements for which *function* is false. For associative collections, the function is passed two arguments (key and value).

filter! (*function*, *collection*)

Update *collection*, removing elements for which *function* is false. For associative collections, the function is passed two arguments (key and value).

2.2.4 Indexable Collections

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, ...)`.

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, ...)`.

Fully implemented by: `Array`, `DArray`, `BitArray`, `AbstractArray`, `SubArray`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `String`.

Partially implemented by: `Range`, `UnitRange`, `Tuple`.

2.2.5 Associative Collections

`Dict` is the standard associative collection. Its implementation uses the `hash(x)` as the hashing function for the key, and `isequal(x, y)` to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

`ObjectIdDict` 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.

Dicts can be created using a literal syntax: `{"A"=>1, "B"=>2}`. Use of curly brackets will create a `Dict` of type `Dict{Any, Any}`. Use of square brackets will attempt to infer type information from the keys and values (i.e. `["A"=>1, "B"=>2]` creates a `Dict{ASCIIString, Int64}`). To explicitly specify types use the syntax: `(KeyType=>ValueType)[...]`. For example, `(ASCIIString=>Int32)["A"=>1, "B"=>2]`.

As with arrays, Dicts may be created with comprehensions. For example, `{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)`.

Dict()

`Dict{K,V}()` constructs a hash

table with keys of type `K` and values of type `V`. The literal syntax is `{"A"=>1, "B"=>2}` for a `Dict{Any,Any}`, or `["A"=>1, "B"=>2]` for a `Dict` of inferred type.

haskey(*collection*, *key*) → `Bool`

Determine whether a collection has a mapping for a given key.

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.

get(*f::Function*, *collection*, *key*)

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using `do` block syntax:

```
get(dict, key) do
    # default value calculated here
    time()
end
```

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`.

get!(*f::Function*, *collection*, *key*)

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using `do` block syntax:

```
get!(dict, key) do
    # default value calculated here
    time()
end
```

getkey(*collection*, *key*, *default*)

Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

delete!(*collection*, *key*)

Delete the mapping for the given key in a collection, and return the collection.

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.

keys(*collection*)

Return an iterator over all keys in a collection. `collect(keys(d))` returns an array of keys.

values(*collection*)

Return an iterator over all values in a collection. `collect(values(d))` returns an array of values.

merge(*collection*, *others...*)

Construct a merged collection from the given collections.

merge! (*collection*, *others...*)

Update collection with pairs from the other collections

sizehint (*s*, *n*)

Suggest that collection *s* reserve capacity for at least *n* elements. This can improve performance.

Fully implemented by: `ObjectIdDict`, `Dict`, `WeakKeyDict`.

Partially implemented by: `IntSet`, `Set`, `EnvHash`, `Array`, `BitArray`.

2.2.6 Set-Like Collections

Set (*[itr]*)

Construct a `Set` of the values generated by the given iterable object, or an empty set. Should be used instead of `IntSet` for sparse integer sets, or for sets of arbitrary objects.

IntSet (*[itr]*)

Construct a sorted set of the integers generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only non-negative integers can be stored. If the set will be sparse (for example holding a single very large integer), use `Set` instead.

union (*s1*, *s2...*)

\cup (*s1*, *s2*)

Construct the union of two or more sets. Maintains order with arrays.

union! (*s*, *iterable*)

Union each element of *iterable* into set *s* in-place.

intersect (*s1*, *s2...*)

\cap (*s1*, *s2*)

Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

setdiff (*s1*, *s2*)

Construct the set of elements in *s1* but not *s2*. Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, `setdiff(set, element)` where *element* is a potential member of *set*, will not work in general.

setdiff! (*s*, *iterable*)

Remove each element of *iterable* from set *s* in-place.

symdiff (*s1*, *s2...*)

Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

symdiff! (*s*, *n*)

`IntSet` *s* is destructively modified to toggle the inclusion of integer *n*.

symdiff! (*s*, *itr*)

For each element in *itr*, destructively toggle its inclusion in set *s*.

symdiff! (*s1*, *s2*)

Construct the symmetric difference of `IntSets` *s1* and *s2*, storing the result in *s1*.

complement (*s*)

Returns the set-complement of `IntSet` *s*.

complement! (*s*)

Mutates `IntSet` *s* into its set-complement.

intersect! (*s1*, *s2*)

Intersects IntSets *s1* and *s2* and overwrites the set *s1* with the result. If needed, *s1* will be expanded to the size of *s2*.

issubset (*A*, *S*) → Bool

\subseteq (*A*, *S*) → Bool

True if *A* is a subset of or equal to *S*.

Fully implemented by: IntSet, Set.

Partially implemented by: Array.

2.2.7 Dequeues

push! (*collection*, *items...*) → *collection*

Insert items at the end of a collection.

pop! (*collection*) → item

Remove the last item in a collection and return it.

unshift! (*collection*, *items...*) → *collection*

Insert items at the beginning of a collection.

shift! (*collection*) → item

Remove the first item in a collection.

insert! (*collection*, *index*, *item*)

Insert an item at the given index.

deleteat! (*collection*, *index*)

Remove the item at the given index, and return the modified collection. Subsequent items are shifted to fill the resulting gap.

deleteat! (*collection*, *itr*)

Remove the items at the indices given by *itr*, and return the modified collection. Subsequent items are shifted to fill the resulting gap. *itr* must be sorted and unique.

splice! (*collection*, *index*[, *replacement*]) → item

Remove the item at the given index, and return the removed item. Subsequent items are shifted down to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

To insert *replacement* before an index *n* without removing any items, use `splice!(collection, n:n-1, replacement)`.

splice! (*collection*, *range*[, *replacement*]) → items

Remove items in the specified index range, and return a collection containing the removed items. Subsequent items are shifted down 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)`.

resize! (*collection*, *n*) → *collection*

Resize collection to contain *n* elements.

append! (*collection*, *items*) → *collection*.

Add the elements of *items* to the end of a collection.

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3
```

prepend! (*collection, items*) → *collection*

Insert the elements of *items* to the beginning of a collection.

```
julia> prepend!([3],[1,2])
3-element Array{Int64,1}:
 1
 2
 3
```

Fully implemented by: `Vector` (aka 1-d Array), `BitVector` (aka 1-d BitArray).

2.2.8 PriorityQueue

The `PriorityQueue` type is available from the `Collections` module. It provides a basic priority queue implementation allowing for arbitrary key and priority types. Multiple identical keys are not permitted, but the priority of existing keys can be changed efficiently.

PriorityQueue{K,V} (*[ord]*)

Construct a new `PriorityQueue`, with keys of type `K` and values/priorities of type `V`. If an order is not given, the priority queue is min-ordered using the default comparison for `V`.

enqueue! (*pq, k, v*)

Insert the a key *k* into a priority queue *pq* with priority *v*.

dequeue! (*pq*)

Remove and return the lowest priority key from a priority queue.

peek (*pq*)

Return the lowest priority key from a priority queue without removing that key from the queue.

`PriorityQueue` also behaves similarly to a `Dict` so that keys can be inserted and priorities accessed or changed using indexing notation:

```
# Julia code
pq = Collections.PriorityQueue()

# Insert keys with associated priorities
pq["a"] = 10
pq["b"] = 5
pq["c"] = 15

# Change the priority of an existing key
pq["a"] = 0
```

2.2.9 Heap Functions

Along with the `PriorityQueue` type, the `Collections` module provides lower level functions for performing binary heap operations on arrays. Each function takes an optional ordering argument. If not given, default ordering is used, so that elements popped from the heap are given in ascending order.

heapify (*v[, ord]*)

Return a new vector in binary heap order, optionally using the given ordering.

heapify! ($v[, ord]$)

In-place heapify.

isheap ($v[, ord]$)

Return true iff an array is heap-ordered according to the given order.

heappush! ($v, x[, ord]$)

Given a binary heap-ordered array, push a new element x , preserving the heap property. For efficiency, this function does not check that the array is indeed heap-ordered.

heappop! ($v[, ord]$)

Given a binary heap-ordered array, remove and return the lowest ordered element. For efficiency, this function does not check that the array is indeed heap-ordered.

2.3 Mathematics

2.3.1 Mathematical Operators

$- (x)$

Unary minus operator.

$+ (x, y...)$

Addition operator. $x+y+z+...$ calls this function with all arguments, i.e. $+(x, y, z, ...)$.

$- (x, y)$

Subtraction operator.

$* (x, y...)$

Multiplication operator. $x*y*z*...$ calls this function with all arguments, i.e. $*(x, y, z, ...)$.

$/ (x, y)$

Right division operator: multiplication of x by the inverse of y on the right. Gives floating-point results for integer arguments.

$\backslash (x, y)$

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

$^ (x, y)$

Exponentiation operator.

$.+ (x, y)$

Element-wise addition operator.

$.- (x, y)$

Element-wise subtraction operator.

$.* (x, y)$

Element-wise multiplication operator.

$./ (x, y)$

Element-wise right division operator.

$.\backslash (x, y)$

Element-wise left division operator.

$.^ (x, y)$

Element-wise exponentiation operator.

div (a, b)

$\div (a, b)$

Compute a/b , truncating to an integer.

fld (a, b)

Largest integer less than or equal to a/b .

mod (x, m)

Modulus after division, returning in the range $[0, m)$.

mod2pi (x)

Modulus after division by 2π , returning in the range $[0, 2\pi)$.

This function computes a floating point representation of the modulus after division by numerically exact 2π , and is therefore not exactly the same as $\text{mod}(x, 2\pi)$, which would compute the modulus of x relative to division by the floating-point number 2π .

rem (x, m)

Remainder after division.

divrem (x, y)

Returns $(x/y, x\%y)$.

% (x, m)

Remainder after division. The operator form of **rem**.

mod1 (x, m)

Modulus after division, returning in the range $(0, m]$

rem1 (x, m)

Remainder after division, returning in the range $(0, m]$

// (num, den)

Divide two integers or rational numbers, giving a `Rational` result.

rationalize $([Type=Int], x; tol=eps(x))$

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 .

num (x)

Numerator of the rational representation of x

den (x)

Denominator of the rational representation of x

<< (x, n)

Left bit shift operator.

>> (x, n)

Right bit shift operator, preserving the sign of x .

>>> (x, n)

Unsigned right bit shift operator.

: $(start[step], stop)$

Range operator. $a:b$ constructs a range from a to b with a step size of 1, and $a:s:b$ is similar but uses a step size of s . These syntaxes call the function `colon`. The colon is also used in indexing to select whole dimensions.

colon $(start[step], stop)$

Called by `:` syntax for constructing ranges.

range $(start[step], length)$

Construct a range by length, given a starting value and optional step (defaults to 1).

linrange (*start, end, length*)

Construct a range by length, given a starting and ending value.

== (*x, y*)

Generic equality operator, giving a single `Bool` result. 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.

Follows IEEE semantics for floating-point numbers.

Collections should generally implement `==` by calling `==` recursively on all contents.

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.

!= (*x, y*)

≠ (*x, y*)

Not-equals comparison operator. Always gives the opposite answer as `==`. New types should generally not implement this, and rely on the fallback definition `!=(x, y) = !(x==y)` instead.

=== (*x, y*)

≡ (*x, y*)

See the `is()` operator

!== (*x, y*)

≠ (*x, y*)

Equivalent to `!is(x, y)`

< (*x, y*)

Less-than comparison operator. New numeric types should implement this function for two arguments of the new type. Because of the behavior of floating-point NaN values, `<` implements a partial order. Types with a canonical partial order should implement `<`, and types with a canonical total order should implement `isless`.

<= (*x, y*)

≤ (*x, y*)

Less-than-or-equals comparison operator.

> (*x, y*)

Greater-than comparison operator. Generally, new types should implement `<` instead of this function, and rely on the fallback definition `>(x, y) = y<x`.

>= (*x, y*)

≥ (*x, y*)

Greater-than-or-equals comparison operator.

.== (*x, y*)

Element-wise equality comparison operator.

.!= (*x, y*)

.≠ (*x, y*)

Element-wise not-equals comparison operator.

.< (*x, y*)

Element-wise less-than comparison operator.

.<= (*x, y*)

.≤ (*x, y*)

Element-wise less-than-or-equals comparison operator.

.> (*x, y*)

Element-wise greater-than comparison operator.

`.>= (x, y)`

`.≥ (x, y)`

Element-wise greater-than-or-equals comparison operator.

`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`. For floating-point numbers, uses `<` but throws an error for unordered arguments.

`~ (x)`

Bitwise not

`& (x, y)`

Bitwise and

`| (x, y)`

Bitwise or

`$ (x, y)`

Bitwise exclusive or

`! (x)`

Boolean not

`x && y`

Short-circuiting boolean and

`x || y`

Short-circuiting boolean or

`A_ldiv_Bc (a, b)`

Matrix operator $A \setminus B^H$

`A_ldiv_Bt (a, b)`

Matrix operator $A \setminus B^T$

`A_mul_B! (Y, A, B) → Y`

Calculates the matrix-matrix or matrix-vector product $A B$ and stores the result in Y , overwriting the existing value of Y .

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; A_mul_B!(B, A, B);
```

```
julia> B
2x2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

`A_mul_Bc (...)`

Matrix operator $A B^H$

`A_mul_Bt (...)`

Matrix operator $A B^T$

`A_rdiv_Bc (...)`

Matrix operator A / B^H

`A_rdiv_Bt (a, b)`

Matrix operator A / B^T

`Ac_ldiv_B (...)`

Matrix operator $A^H \setminus B$

Ac_ldiv_Bc (...)
 Matrix operator $A^H \setminus B^H$

Ac_mul_B (...)
 Matrix operator $A^H B$

Ac_mul_Bc (...)
 Matrix operator $A^H B^H$

Ac_rdiv_B (a, b)
 Matrix operator A^H / B

Ac_rdiv_Bc (a, b)
 Matrix operator A^H / B^H

At_ldiv_B (...)
 Matrix operator $A^T \setminus B$

At_ldiv_Bt (...)
 Matrix operator $A^T \setminus B^T$

At_mul_B (...)
 Matrix operator $A^T B$

At_mul_Bt (...)
 Matrix operator $A^T B^T$

At_rdiv_B (a, b)
 Matrix operator A^T / B

At_rdiv_Bt (a, b)
 Matrix operator A^T / B^T

2.3.2 Mathematical Functions

isapprox (x::Number, y::Number; rtol::Real=cbrt(maxeps), atol::Real=sqrt(maxeps))

Inexact equality comparison - behaves slightly different depending on types of input args:

- For `FloatingPoint` numbers, `isapprox` returns true if `abs(x-y) <= atol + rtol*max(abs(x), abs(y))`.
- For `Integer` and `Rational` numbers, `isapprox` returns true if `abs(x-y) <= atol`. The `rtol` argument is ignored. If one of `x` and `y` is `FloatingPoint`, the other is promoted, and the method above is called instead.
- For `Complex` numbers, the distance in the complex plane is compared, using the same criterion as above.

For default tolerance arguments, `maxeps = max(eps(abs(x)), eps(abs(y)))`.

sin (x)
 Compute sine of x, where x is in radians

cos (x)
 Compute cosine of x, where x is in radians

tan (x)
 Compute tangent of x, where x is in radians

sind (x)
 Compute sine of x, where x is in degrees

cosd (x)
 Compute cosine of x, where x is in degrees

tand(*x*)

Compute tangent of *x*, where *x* is in degrees

sinpi(*x*)

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large *x*.

cospi(*x*)

Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large *x*.

sinh(*x*)

Compute hyperbolic sine of *x*

cosh(*x*)

Compute hyperbolic cosine of *x*

tanh(*x*)

Compute hyperbolic tangent of *x*

asin(*x*)

Compute the inverse sine of *x*, where the output is in radians

acos(*x*)

Compute the inverse cosine of *x*, where the output is in radians

atan(*x*)

Compute the inverse tangent of *x*, where the output is in radians

atan2(*y*, *x*)

Compute the inverse tangent of *y*/*x*, using the signs of both *x* and *y* to determine the quadrant of the return value.

asind(*x*)

Compute the inverse sine of *x*, where the output is in degrees

acosd(*x*)

Compute the inverse cosine of *x*, where the output is in degrees

atand(*x*)

Compute the inverse tangent of *x*, where the output is in degrees

sec(*x*)

Compute the secant of *x*, where *x* is in radians

csc(*x*)

Compute the cosecant of *x*, where *x* is in radians

cot(*x*)

Compute the cotangent of *x*, where *x* is in radians

secd(*x*)

Compute the secant of *x*, where *x* is in degrees

cscd(*x*)

Compute the cosecant of *x*, where *x* is in degrees

cotd(*x*)

Compute the cotangent of *x*, where *x* is in degrees

asec(*x*)

Compute the inverse secant of *x*, where the output is in radians

acsc(*x*)

Compute the inverse cosecant of *x*, where the output is in radians

acot (*x*)

Compute the inverse cotangent of *x*, where the output is in radians

asecd (*x*)

Compute the inverse secant of *x*, where the output is in degrees

acscd (*x*)

Compute the inverse cosecant of *x*, where the output is in degrees

acotd (*x*)

Compute the inverse cotangent of *x*, where the output is in degrees

sech (*x*)

Compute the hyperbolic secant of *x*

csch (*x*)

Compute the hyperbolic cosecant of *x*

coth (*x*)

Compute the hyperbolic cotangent of *x*

asinh (*x*)

Compute the inverse hyperbolic sine of *x*

acosh (*x*)

Compute the inverse hyperbolic cosine of *x*

atanh (*x*)

Compute the inverse hyperbolic tangent of *x*

asech (*x*)

Compute the inverse hyperbolic secant of *x*

acsch (*x*)

Compute the inverse hyperbolic cosecant of *x*

acoth (*x*)

Compute the inverse hyperbolic cotangent of *x*

sinc (*x*)

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

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*).

deg2rad (*x*)

Convert *x* from degrees to radians

rad2deg (*x*)

Convert *x* from radians to degrees

hypot (*x*, *y*)

Compute the $\sqrt{x^2 + y^2}$ avoiding overflow and underflow

log (*x*)

Compute the natural logarithm of *x*. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead.

log (*b*, *x*)

Compute the base *b* logarithm of *x*. Throws `DomainError` for negative `Real` arguments.

log2 (*x*)

Compute the logarithm of *x* to base 2. Throws `DomainError` for negative `Real` arguments.

log10(*x*)
 Compute the logarithm of *x* to base 10. Throws `DomainError` for negative `Real` arguments.

log1p(*x*)
 Accurate natural logarithm of $1+x$. Throws `DomainError` for `Real` arguments less than -1.

frexp(*val*)
 Return (*x*, *exp*) such that *x* has a magnitude in the interval $[1/2, 1)$ or 0, and $val = x \times 2^{exp}$.

exp(*x*)
 Compute e^x

exp2(*x*)
 Compute 2^x

exp10(*x*)
 Compute 10^x

ldexp(*x*, *n*)
 Compute $x \times 2^n$

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.

expm1(*x*)
 Accurately compute $e^x - 1$

round(*x*[, *digits*[, *base*]])
round(*x*) returns the nearest integral value of the same type as *x* to *x*. *round*(*x*, *digits*) rounds to the specified number of digits after the decimal place, or before if negative, e.g., *round*(`pi`, 2) is 3.14. *round*(*x*, *digits*, *base*) rounds using a different base, defaulting to 10, e.g., *round*(`pi`, 1, 8) is 3.125.

ceil(*x*[, *digits*[, *base*]])
 Returns the nearest integral value of the same type as *x* not less than *x*. *digits* and *base* work as above.

floor(*x*[, *digits*[, *base*]])
 Returns the nearest integral value of the same type as *x* not greater than *x*. *digits* and *base* work as above.

trunc(*x*[, *digits*[, *base*]])
 Returns the nearest integral value of the same type as *x* not greater in magnitude than *x*. *digits* and *base* work as above.

iround(*x*) → Integer
 Returns the nearest integer to *x*.

iceil(*x*) → Integer
 Returns the nearest integer not less than *x*.

ifloor(*x*) → Integer
 Returns the nearest integer not greater than *x*.

itrunc(*x*) → Integer
 Returns the nearest integer not greater in magnitude than *x*.

signif(*x*, *digits*[, *base*])
 Rounds (in the sense of *round*) *x* so that there are *digits* significant digits, under a base *base* representation, default 10. E.g., *signif*(123.456, 2) is 120.0, and *signif*(357.913, 4, 2) is 352.0.

min(*x*, *y*, ...)
 Return the minimum of the arguments. Operates elementwise over arrays.

max(*x*, *y*, ...)

Return the maximum of the arguments. Operates elementwise over arrays.

minmax(*x*, *y*)

Return $(\min(x, y), \max(x, y))$. See also: `extrema()` that returns $(\min(x), \max(x))$

clamp(*x*, *lo*, *hi*)

Return *x* if $lo \leq x \leq hi$. If $x < lo$, return *lo*. If $x > hi$, return *hi*. Arguments are promoted to a common type. Operates elementwise over *x* if it is an array.

abs(*x*)

Absolute value of *x*

abs2(*x*)

Squared absolute value of *x*

copysign(*x*, *y*)

Return *x* such that it has the same sign as *y*

sign(*x*)

Return +1 if *x* is positive, 0 if $x == 0$, and -1 if *x* is negative.

signbit(*x*)

Returns `true` if the value of the sign of *x* is negative, otherwise `false`.

flipsign(*x*, *y*)

Return *x* with its sign flipped if *y* is negative. For example $\text{abs}(x) = \text{flipsign}(x, x)$.

sqrt(*x*)

Return \sqrt{x} . Throws `DomainError` for negative Real arguments. Use complex negative arguments instead. The prefix operator $\sqrt{}$ is equivalent to `sqrt`.

isqrt(*n*)

Integer square root: the largest integer *m* such that $m*m \leq n$.

cbrt(*x*)

Return $x^{1/3}$. The prefix operator $\sqrt[3]{}$ is equivalent to `cbrt`.

erf(*x*)

Compute the error function of *x*, defined by $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ for arbitrary complex *x*.

erfc(*x*)

Compute the complementary error function of *x*, defined by $1 - \text{erf}(x)$.

erfcx(*x*)

Compute the scaled complementary error function of *x*, defined by $e^{x^2} \text{erfc}(x)$. Note also that `erfcx(-ix)` computes the Faddeeva function $w(x)$.

erfi(*x*)

Compute the imaginary error function of *x*, defined by $-i \text{erf}(ix)$.

dawson(*x*)

Compute the Dawson function (scaled imaginary error function) of *x*, defined by $\frac{\sqrt{\pi}}{2} e^{-x^2} \text{erfi}(x)$.

erfinv(*x*)

Compute the inverse error function of a real *x*, defined by $\text{erf}(\text{erfinv}(x)) = x$.

erfcinv(*x*)

Compute the inverse error complementary function of a real *x*, defined by $\text{erfc}(\text{erfcinv}(x)) = x$.

real(*z*)

Return the real part of the complex number *z*

imag(*z*)Return the imaginary part of the complex number *z***reim**(*z*)Return both the real and imaginary parts of the complex number *z***conj**(*z*)Compute the complex conjugate of a complex number *z***angle**(*z*)Compute the phase angle of a complex number *z***cis**(*z*)Return $\exp(iz)$.**binomial**(*n*, *k*)Number of ways to choose *k* out of *n* items**factorial**(*n*)Factorial of *n***factorial**(*n*, *k*)Compute $\text{factorial}(n) / \text{factorial}(k)$ **factor**(*n*) → Dict

Compute the prime factorization of an integer *n*. Returns a dictionary. The keys of the dictionary correspond to the factors, and hence are of the same type as *n*. The value associated with each key indicates the number of times the factor appears in the factorization.

```
julia> factor(100) # == 2*2*5*5
Dict{Int64, Int64} with 2 entries:
 2 => 2
 5 => 2
```

gcd(*x*, *y*)Greatest common (positive) divisor (or zero if *x* and *y* are both zero).**lcm**(*x*, *y*)

Least common (non-negative) multiple.

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 $ux + vy = d = \text{gcd}(x, y)$.

```
julia> gcdx(12, 42)
(6, -3, 1)
```

```
julia> gcdx(240, 46)
(2, -9, 47)
```

Note: Bézout coefficients are *not* uniquely defined. `gcdx` returns the minimal Bézout coefficients that are computed by the extended Euclid algorithm. (Ref: D. Knuth, TAOCP, 2/e, p. 325, Algorithm X.) These coefficients *u* and *v* are minimal in the sense that $|u| < |\frac{y}{d}|$ and $|v| < |\frac{x}{d}|$. Furthermore, the signs of *u* and *v* are chosen so that *d* is positive.

ispow2(*n*) → BoolTest whether *n* is a power of two**nextpow2**(*n*)

The smallest power of two not less than *n*. Returns 0 for *n*==0, and returns `-nextpow2(-n)` for negative arguments.

prevpow2(*n*)

The largest power of two not greater than *n*. Returns 0 for *n*==0, and returns `-prevpow2(-n)` for negative arguments.

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.

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.

nextprod(*[k_1, k_2, ...]*, *n*)

Next integer not less than *n* that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2 , etc.

prevprod(*[k_1, k_2, ...]*, *n*)

Previous integer not greater than *n* that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2 , etc.

invmod(*x*, *m*)

Take the inverse of *x* modulo *m*: *y* such that $xy = 1 \pmod{m}$

powermod(*x*, *p*, *m*)

Compute $x^p \pmod{m}$

gamma(*x*)

Compute the gamma function of *x*

lgamma(*x*)

Compute the logarithm of the absolute value of `gamma(x)` for `Real()` *x*, while for `Complex()` *x* it computes the logarithm of `gamma(x)`.

lfact(*x*)

Compute the logarithmic factorial of *x*

digamma(*x*)

Compute the digamma function of *x* (the logarithmic derivative of `gamma(x)`)

invdigamma(*x*)

Compute the inverse digamma function of *x*.

trigamma(*x*)

Compute the trigamma function of *x* (the logarithmic second derivative of `gamma(x)`)

polygamma(*m*, *x*)

Compute the polygamma function of order *m* of argument *x* (the $(m+1)$ th derivative of the logarithm of `gamma(x)`)

airy(*k*, *x*)

*k*th derivative of the Airy function $\text{Ai}(x)$.

airyai(*x*)

Airy function $\text{Ai}(x)$.

airyprime(*x*)

Airy function derivative $\text{Ai}'(x)$.

airyaiprime(*x*)

Airy function derivative $\text{Ai}'(x)$.

airybi(*x*)

Airy function $\text{Bi}(x)$.

airybiprime (x)

Airy function derivative $\text{Bi}'(x)$.

airyx (k, x)

scaled k th derivative of the Airy function, return $\text{Ai}(x)e^{\frac{2}{3}x\sqrt{x}}$ for $k == 0 \ || \ k == 1$, and $\text{Ai}(x)e^{-|\text{Re}(\frac{2}{3}x\sqrt{x})|}$ for $k == 2 \ || \ k == 3$.

besselj0 (x)

Bessel function of the first kind of order 0, $J_0(x)$.

besselj1 (x)

Bessel function of the first kind of order 1, $J_1(x)$.

besselj (ν, x)

Bessel function of the first kind of order ν , $J_\nu(x)$.

besseljx (ν, x)

Scaled Bessel function of the first kind of order ν , $J_\nu(x)e^{-|\text{Im}(x)|}$.

bessely0 (x)

Bessel function of the second kind of order 0, $Y_0(x)$.

bessely1 (x)

Bessel function of the second kind of order 1, $Y_1(x)$.

bessely (ν, x)

Bessel function of the second kind of order ν , $Y_\nu(x)$.

besselyx (ν, x)

Scaled Bessel function of the second kind of order ν , $Y_\nu(x)e^{-|\text{Im}(x)|}$.

hankelh1 (ν, x)

Bessel function of the third kind of order ν , $H_\nu^{(1)}(x)$.

hankelh1x (ν, x)

Scaled Bessel function of the third kind of order ν , $H_\nu^{(1)}(x)e^{-xi}$.

hankelh2 (ν, x)

Bessel function of the third kind of order ν , $H_\nu^{(2)}(x)$.

hankelh2x (ν, x)

Scaled Bessel function of the third kind of order ν , $H_\nu^{(2)}(x)e^{xi}$.

besselh (ν, k, x)

Bessel function of the third kind of order ν (Hankel function). k is either 1 or 2, selecting **hankelh1** or **hankelh2**, respectively.

besseli (ν, x)

Modified Bessel function of the first kind of order ν , $I_\nu(x)$.

besselix (ν, x)

Scaled modified Bessel function of the first kind of order ν , $I_\nu(x)e^{-|\text{Re}(x)|}$.

besselk (ν, x)

Modified Bessel function of the second kind of order ν , $K_\nu(x)$.

besselkx (ν, x)

Scaled modified Bessel function of the second kind of order ν , $K_\nu(x)e^x$.

beta (x, y)

Euler integral of the first kind $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x + y)$.

lbeta (x, y)

Natural logarithm of the absolute value of the beta function $\log(|B(x, y)|)$.

eta(*x*)

Dirichlet eta function $\eta(s) = \sum_{n=1}^{\infty} (-1)^{n-1} / n^s$.

zeta(*s*)

Riemann zeta function $\zeta(s)$.

zeta(*s*, *z*)

Hurwitz zeta function $\zeta(s, z)$. (This is equivalent to the Riemann zeta function $\zeta(s)$ for the case of $z=1$.)

ndigits(*n*, *b*)

Compute the number of digits in number *n* written in base *b*.

widemul(*x*, *y*)

Multiply *x* and *y*, giving the result as a larger type.

@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.

2.3.3 Statistics

mean(*v*[, *region*])

Compute the mean of whole array *v*, or optionally along the dimensions in *region*. Note: Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

mean!(*r*, *v*)

Compute the mean of *v* over the singleton dimensions of *r*, and write results to *r*.

std(*v*[, *region*])

Compute the sample standard deviation of a vector or array *v*, optionally along dimensions in *region*. 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 $\sqrt{\text{sum}((v - \text{mean}(v)).^2) / (\text{length}(v) - 1)}$. Note: Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

stdm(*v*, *m*)

Compute the sample standard deviation of a vector *v* with known mean *m*. Note: Julia does not ignore NaN values in the computation.

var(*v*[, *region*])

Compute the sample variance of a vector or array *v*, optionally along dimensions in *region*. 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 $\text{sum}((v - \text{mean}(v)).^2) / (\text{length}(v) - 1)$. Note: Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

varm(*v*, *m*)

Compute the sample variance of a vector *v* with known mean *m*. Note: Julia does not ignore NaN values in the computation.

middle(*x*)

Compute the middle of a scalar value, which is equivalent to *x* itself, but of the type of `middle(x, x)` for consistency.

middle (*x*, *y*)

Compute the middle of two reals *x* and *y*, which is equivalent in both value and type to computing their mean $((x + y) / 2)$.

middle (*range*)

Compute the middle of a range, which consists in computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

middle (*array*)

Compute the middle of an array, which consists in finding its extrema and then computing their mean.

median (*v*; *checknan::Bool=true*)

Compute the median of a vector *v*. If the keyword argument *checknan* is true (the default), NaN is returned for data containing NaN values. Otherwise the median is computed with NaN values sorted to the last position. For applications requiring the handling of missing data, the `DataArrays` package is recommended.

median! (*v*; *checknan::Bool=true*)

Like `median`, but may overwrite the input vector.

hist (*v*[, *n*]) → *e*, *counts*

Compute the histogram of *v*, optionally using approximately *n* bins. The return values are a range *e*, which correspond to the edges of the bins, and *counts* containing the number of elements of *v* in each bin. Note: Julia does not ignore NaN values in the computation.

hist (*v*, *e*) → *e*, *counts*

Compute the histogram of *v* using a vector/range *e* as the edges for the bins. The result will be a vector of length `length(e) - 1`, such that the element at location *i* satisfies `sum(e[i] .< v .<= e[i+1])`. Note: Julia does not ignore NaN values in the computation.

hist! (*counts*, *v*, *e*) → *e*, *counts*

Compute the histogram of *v*, using a vector/range *e* as the edges for the bins. This function writes the resultant counts to a pre-allocated array *counts*.

hist2d (*M*, *e1*, *e2*) → (*edge1*, *edge2*, *counts*)

Compute a “2d histogram” of a set of *N* points specified by *N*-by-2 matrix *M*. Arguments *e1* and *e2* are bins for each dimension, specified either as integer bin counts or vectors of bin edges. The result is a tuple of *edge1* (the bin edges used in the first dimension), *edge2* (the bin edges used in the second dimension), and *counts*, a histogram matrix of size `(length(edge1)-1, length(edge2)-1)`. Note: Julia does not ignore NaN values in the computation.

hist2d! (*counts*, *M*, *e1*, *e2*) → (*e1*, *e2*, *counts*)

Compute a “2d histogram” with respect to the bins delimited by the edges given in *e1* and *e2*. This function writes the results to a pre-allocated array *counts*.

histrange (*v*, *n*)

Compute *nice* bin ranges for the edges of a histogram of *v*, using approximately *n* bins. The resulting step sizes will be 1, 2 or 5 multiplied by a power of 10. Note: Julia does not ignore NaN values in the computation.

midpoints (*e*)

Compute the midpoints of the bins with edges *e*. The result is a vector/range of length `length(e) - 1`. Note: Julia does not ignore NaN values in the computation.

quantile (*v*, *p*)

Compute the quantiles of a vector *v* at a specified set of probability values *p*. Note: Julia does not ignore NaN values in the computation.

quantile (*v*, *p*)

Compute the quantile of a vector *v* at the probability *p*. Note: Julia does not ignore NaN values in the computation.

quantile! (*v*, *p*)

Like `quantile`, but overwrites the input vector.

cov (*v1*[], *v2*[], *vardim*=1, *corrected*=true, *mean*=nothing])

Compute the Pearson covariance between the vector(s) in *v1* and *v2*. Here, *v1* and *v2* can be either vectors or matrices.

This function accepts three keyword arguments:

- **vardim**: the dimension of variables. When *vardim* = 1, variables are considered in columns while observations in rows; when *vardim* = 2, variables are in rows while observations in columns. By default, it is set to 1.
- **corrected**: whether to apply Bessel's correction (divide by *n*-1 instead of *n*). By default, it is set to true.
- **mean**: allow users to supply mean values that are known. By default, it is set to `nothing`, which indicates that the mean(s) are unknown, and the function will compute the mean. Users can use *mean*=0 to indicate that the input data are centered, and hence there's no need to subtract the mean.

The size of the result depends on the size of *v1* and *v2*. When both *v1* and *v2* are vectors, it returns the covariance between them as a scalar. When either one is a matrix, it returns a covariance matrix of size (*n1*, *n2*), where *n1* and *n2* are the numbers of slices in *v1* and *v2*, which depend on the setting of *vardim*.

Note: *v2* can be omitted, which indicates *v2* = *v1*.

cor (*v1*[], *v2*[], *vardim*=1, *mean*=nothing])

Compute the Pearson correlation between the vector(s) in *v1* and *v2*.

Users can use the keyword argument *vardim* to specify the variable dimension, and *mean* to supply pre-computed mean values.

2.3.4 Signal Processing

Fast Fourier transform (FFT) functions in Julia are largely implemented by calling functions from `FFTW`. By default, Julia does not use multi-threaded FFTW. Higher performance may be obtained by experimenting with multi-threading. Use `FFTW.set_num_threads(np)` to use *np* threads.

fft (*A*[], *dims*[])

Performs a multidimensional FFT of the array *A*. The optional *dims* argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of *A* along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_fft()` for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

$$\text{DFT}(A)[k] = \sum_{n=1}^{\text{length}(A)} \exp\left(-i \frac{2\pi(n-1)(k-1)}{\text{length}(A)}\right) A[n].$$

A multidimensional FFT simply performs this operation along each transformed dimension of *A*.

Higher performance is usually possible with multi-threading. Use `FFTW.set_num_threads(np)` to use *np* threads, if you have *np* processors.

fft! (*A*[], *dims*[])

Same as `fft()`, but operates in-place on *A*, which must be an array of complex floating-point numbers.

ifft (*A*[], *dims*[])

Multidimensional inverse FFT.

A one-dimensional inverse FFT computes

$$\text{IDFT}(A)[k] = \frac{1}{\text{length}(A)} \sum_{n=1}^{\text{length}(A)} \exp\left(+i \frac{2\pi(n-1)(k-1)}{\text{length}(A)}\right) A[n].$$

A multidimensional inverse FFT simply performs this operation along each transformed dimension of `A`.

ifft! (`A`[, `dims`])

Same as `ifft()`, but operates in-place on `A`.

bfft (`A`[, `dims`])

Similar to `ifft()`, but computes an unnormalized inverse (backward) transform, which must be divided by the product of the sizes of the transformed dimensions in order to obtain the inverse. (This is slightly more efficient than `ifft()` because it omits a scaling step, which in some applications can be combined with other computational steps elsewhere.)

$$\text{BDFT}(A)[k] = \text{length}(A) \text{IDFT}(A)[k]$$

bfft! (`A`[, `dims`])

Same as `bfft()`, but operates in-place on `A`.

plan_fft (`A`[, `dims`[, `flags`[, `timelimit`]]])

Pre-plan an optimized FFT along given dimensions (`dims`) of arrays matching the shape and type of `A`. (The first two arguments have the same meaning as for `fft()`.) Returns a function `plan(A)` that computes `fft(A, dims)` quickly.

The `flags` argument is a bitwise-or of FFTW planner flags, defaulting to `FFTW.ESTIMATE`. e.g. passing `FFTW.MEASURE` or `FFTW.PATIENT` will instead spend several seconds (or more) benchmarking different possible FFT algorithms and picking the fastest one; see the FFTW manual for more information on planner flags. The optional `timelimit` argument specifies a rough upper bound on the allowed planning time, in seconds. Passing `FFTW.MEASURE` or `FFTW.PATIENT` may cause the input array `A` to be overwritten with zeros during plan creation.

`plan_fft!()` is the same as `plan_fft()` but creates a plan that operates in-place on its argument (which must be an array of complex floating-point numbers). `plan_ifft()` and so on are similar but produce plans that perform the equivalent of the inverse transforms `ifft()` and so on.

plan_ifft (`A`[, `dims`[, `flags`[, `timelimit`]]])

Same as `plan_fft()`, but produces a plan that performs inverse transforms `ifft()`.

plan_bfft (`A`[, `dims`[, `flags`[, `timelimit`]]])

Same as `plan_fft()`, but produces a plan that performs an unnormalized backwards transform `bfft()`.

plan_fft! (`A`[, `dims`[, `flags`[, `timelimit`]]])

Same as `plan_fft()`, but operates in-place on `A`.

plan_ifft! (`A`[, `dims`[, `flags`[, `timelimit`]]])

Same as `plan_ifft()`, but operates in-place on `A`.

plan_bfft! (`A`[, `dims`[, `flags`[, `timelimit`]]])

Same as `plan_bfft()`, but operates in-place on `A`.

rfft (`A`[, `dims`])

Multidimensional FFT of a real array `A`, exploiting the fact that the transform has conjugate symmetry in order to save roughly half the computational time and storage costs compared with `fft()`. If `A` has size `(n_1, ..., n_d)`, the result has size `(floor(n_1/2)+1, ..., n_d)`.

The optional `dims` argument specifies an iterable subset of one or more dimensions of `A` to transform, similar to `fft()`. Instead of (roughly) halving the first dimension of `A` in the result, the `dims[1]` dimension is (roughly) halved in the same way.

irfft (*A*, *d*[, *dims*])

Inverse of `rfft()`: for a complex array *A*, gives the corresponding real array whose FFT yields *A* in the first half. As for `rfft()`, *dims* is an optional subset of dimensions to transform, defaulting to `1:ndims(A)`.

d is the length of the transformed real array along the `dims[1]` dimension, which must satisfy `d == floor(size(A, dims[1])/2)+1`. (This parameter cannot be inferred from `size(A)` due to the possibility of rounding by the `floor` function here.)

brfft (*A*, *d*[, *dims*])

Similar to `irfft()` but computes an unnormalized inverse transform (similar to `bfft()`), which must be divided by the product of the sizes of the transformed dimensions (of the real output array) in order to obtain the inverse transform.

plan_rfft (*A*[, *dims*[, *flags*[, *timelimit*]]])

Pre-plan an optimized real-input FFT, similar to `plan_fft()` except for `rfft()` instead of `fft()`. The first two arguments, and the size of the transformed result, are the same as for `rfft()`.

plan_brfft (*A*, *d*[, *dims*[, *flags*[, *timelimit*]]])

Pre-plan an optimized real-input unnormalized transform, similar to `plan_rfft()` except for `brfft()` instead of `rfft()`. The first two arguments and the size of the transformed result, are the same as for `brfft()`.

plan_irfft (*A*, *d*[, *dims*[, *flags*[, *timelimit*]]])

Pre-plan an optimized inverse real-input FFT, similar to `plan_rfft()` except for `irfft()` and `brfft()`, respectively. The first three arguments have the same meaning as for `irfft()`.

dct (*A*[, *dims*])

Performs a multidimensional type-II discrete cosine transform (DCT) of the array *A*, using the unitary normalization of the DCT. The optional *dims* argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of *A* along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_dct()` for even greater efficiency.

dct! (*A*[, *dims*])

Same as `dct()`, except that it operates in-place on *A*, which must be an array of real or complex floating-point values.

idct (*A*[, *dims*])

Computes the multidimensional inverse discrete cosine transform (DCT) of the array *A* (technically, a type-III DCT with the unitary normalization). The optional *dims* argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of *A* along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_idct()` for even greater efficiency.

idct! (*A*[, *dims*])

Same as `idct()`, but operates in-place on *A*.

plan_dct (*A*[, *dims*[, *flags*[, *timelimit*]]])

Pre-plan an optimized discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `dct()`. The first two arguments have the same meaning as for `dct()`.

plan_dct! (*A*[, *dims*[, *flags*[, *timelimit*]]])

Same as `plan_dct()`, but operates in-place on *A*.

plan_idct (*A*[, *dims*[, *flags*[, *timelimit*]]])

Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `idct()`. The first two arguments have the same meaning as for `idct()`.

plan_idct! (*A*[, *dims*[, *flags*[, *timelimit*]]])

Same as `plan_idct()`, but operates in-place on *A*.

fftshift (*x*)

Swap the first and second halves of each dimension of *x*.

fftshift (*x*, *dim*)

Swap the first and second halves of the given dimension of array *x*.

ifftshift (*x*[:, *dim*])

Undoes the effect of `fftshift`.

filt (*b*, *a*, *x*[:, *si*])

Apply filter described by vectors *a* and *b* to vector *x*, with an optional initial filter state vector *si* (defaults to zeros).

filt! (*out*, *b*, *a*, *x*[:, *si*])

Same as `filt()` but writes the result into the *out* argument, which may alias the input *x* to modify it in-place.

deconv (*b*, *a*)

Construct vector *c* such that $b = \text{conv}(a, c) + r$. Equivalent to polynomial division.

conv (*u*, *v*)

Convolution of two vectors. Uses FFT algorithm.

conv2 (*u*, *v*, *A*)

2-D convolution of the matrix *A* with the 2-D separable kernel generated by the vectors *u* and *v*. Uses 2-D FFT algorithm

conv2 (*B*, *A*)

2-D convolution of the matrix *B* with the matrix *A*. Uses 2-D FFT algorithm

xcorr (*u*, *v*)

Compute the cross-correlation of two vectors.

The following functions are defined within the `Base.FFTW` module.

r2r (*A*, *kind*[:, *dims*])

Performs a multidimensional real-input/real-output (r2r) transform of type *kind* of the array *A*, as defined in the FFTW manual. *kind* specifies either a discrete cosine transform of various types (`FFTW.REDFT00`, `FFTW.REDFT01`, `FFTW.RODFT00`, `FFTW.RODFT01`, `FFTW.RODFT10`, or `FFTW.RODFT11`), a discrete sine transform of various types (`FFTW.REDFT00`, `FFTW.RODFT01`, `FFTW.RODFT10`, or `FFTW.RODFT11`), a real-input DFT with halfcomplex-format output (`FFTW.R2HC` and its inverse `FFTW.HC2R`), or a discrete Hartley transform (`FFTW.DHT`). The *kind* argument may be an array or tuple in order to specify different transform types along the different dimensions of *A*; *kind*[*end*] is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at <http://www.fftw.org/doc>.

The optional *dims* argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. *kind*[*i*] is then the transform type for *dims*[*i*], with *kind*[*end*] being used for $i > \text{length}(\text{kind})$.

See also `plan_r2r()` to pre-plan optimized r2r transforms.

r2r! (*A*, *kind*[:, *dims*])

Same as `r2r()`, but operates in-place on *A*, which must be an array of real or complex floating-point numbers.

plan_r2r (*A*, *kind*[:, *dims*[:, *flags*[:, *timelimit*]]])

Pre-plan an optimized r2r transform, similar to `Base.plan_fft()` except that the transforms (and the first three arguments) correspond to `r2r()` and `r2r!()`, respectively.

plan_r2r! (*A*, *kind*[:, *dims*[:, *flags*[:, *timelimit*]]])

Similar to `Base.plan_fft()`, but corresponds to `r2r!()`.

2.3.5 Numerical Integration

Although several external packages are available for numeric integration and solution of ordinary differential equations, we also provide some built-in integration support in Julia.

quadgk (*f*, *a*, *b*, *c*...; *reitol*=*sqrt(eps)*, *abstol*=0, *maxevals*= 10^7 , *order*=7, *norm*=*vecnorm*)

Numerically integrate the function $f(x)$ from *a* to *b*, and optionally over additional intervals *b* to *c* and so on. Keyword options include a relative error tolerance *reitol* (defaults to *sqrt(eps)* in the precision of the endpoints), an absolute error tolerance *abstol* (defaults to 0), a maximum number of function evaluations *maxevals* (defaults to 10^7), and the *order* of the integration rule (defaults to 7).

Returns a pair (*I*, *E*) of the estimated integral *I* and an estimated upper bound on the absolute error *E*. If *maxevals* is not exceeded then $E \leq \max(\text{abstol}, \text{reitol} * \text{norm}(I))$ will hold. (Note that it is useful to specify a positive *abstol* in cases where *norm(I)* may be zero.)

The endpoints *a* etcetera can also be complex (in which case the integral is performed over straight-line segments in the complex plane). If the endpoints are `BigFloat`, then the integration will be performed in `BigFloat` precision as well (note: it is advisable to increase the integration *order* in rough proportion to the precision, for smooth integrands). More generally, the precision is set by the precision of the integration endpoints (promoted to floating-point types).

The integrand $f(x)$ can return any numeric scalar, vector, or matrix type, or in fact any type supporting $+$, $-$, multiplication by real values, and a *norm* (i.e., any normed vector space). Alternatively, a different norm can be specified by passing a *norm*-like function as the *norm* keyword argument (which defaults to *vecnorm*).

The algorithm is an adaptive Gauss-Kronrod integration technique: the integral in each interval is estimated using a Kronrod rule ($2 * \text{order} + 1$ points) and the error is estimated using an embedded Gauss rule (*order* points). The interval with the largest error is then subdivided into two intervals and the process is repeated until the desired error tolerance is achieved.

These quadrature rules work best for smooth functions within each interval, so if your function has a known discontinuity or other singularity, it is best to subdivide your interval to put the singularity at an endpoint. For example, if f has a discontinuity at $x = 0.7$ and you want to integrate from 0 to 1, you should use `quadgk(f, 0, 0.7, 1)` to subdivide the interval at the point of discontinuity. The integrand is never evaluated exactly at the endpoints of the intervals, so it is possible to integrate functions that diverge at the endpoints as long as the singularity is integrable (for example, a $\log(x)$ or $1/\sqrt{x}$ singularity).

For real-valued endpoints, the starting and/or ending points may be infinite. (A coordinate transformation is performed internally to map the infinite interval to a finite one.)

2.4 Numbers

2.4.1 Standard Numeric Types

`Bool` `Int8` `UInt8` `Int16` `UInt16` `Int32` `UInt32` `Int64` `UInt64` `Int128` `UInt128` `Float16` `Float32`
`Float64` `Complex64` `Complex128`

2.4.2 Data Formats

bin (*n*[, *pad*])

Convert an integer to a binary string, optionally specifying a number of digits to pad to.

hex (*n*[, *pad*])

Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

dec (*n*[, *pad*])

Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

oct (*n*[, *pad*])

Convert an integer to an octal string, optionally specifying a number of digits to pad to.

base (*base*, *n* [, *pad*])

Convert an integer to a string in the given base, optionally specifying a number of digits to pad to. The base can be specified as either an integer, or as a `UInt8` array of character values to use as digit symbols.

digits (*n* [, *base*] [, *pad*])

Returns an array of the digits of *n* in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

bits (*n*)

A string giving the literal bit representation of a number.

parseint ([*type*], *str* [, *base*])

Parse a string as an integer in the given base (default 10), yielding a number of the specified type (default `Int`).

parsefloat ([*type*], *str*)

Parse a string as a decimal floating point number, yielding a number of the specified type.

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.

bool (*x*)

Convert a number or numeric array to boolean

int (*x*)

Convert a number or array to the default integer type on your platform. Alternatively, *x* can be a string, which is parsed as an integer.

uint (*x*)

Convert a number or array to the default unsigned integer type on your platform. Alternatively, *x* can be a string, which is parsed as an unsigned integer.

integer (*x*)

Convert a number or array to integer type. If *x* is already of integer type it is unchanged, otherwise it converts it to the default integer type on your platform.

signed (*x*)

Convert a number to a signed integer

unsigned (*x*) \rightarrow `Unsigned`

Convert a number to an unsigned integer

int8 (*x*)

Convert a number or array to `Int8` data type

int16 (*x*)

Convert a number or array to `Int16` data type

int32 (*x*)

Convert a number or array to `Int32` data type

int64 (*x*)

Convert a number or array to `Int64` data type

int128 (*x*)

Convert a number or array to `Int128` data type

uint8 (*x*)

Convert a number or array to `UInt8` data type

uint16 (*x*)

Convert a number or array to `UInt16` data type

uint32(*x*)

Convert a number or array to `UInt32` data type

uint64(*x*)

Convert a number or array to `UInt64` data type

uint128(*x*)

Convert a number or array to `UInt128` data type

float16(*x*)

Convert a number or array to `Float16` data type

float32(*x*)

Convert a number or array to `Float32` data type

float64(*x*)

Convert a number or array to `Float64` data type

float32_isvalid(*x*, *out*::`Vector{Float32}`) → `Bool`

Convert a number or array to `Float32` data type, returning true if successful. The result of the conversion is stored in `out[1]`.

float64_isvalid(*x*, *out*::`Vector{Float64}`) → `Bool`

Convert a number or array to `Float64` data type, returning true if successful. The result of the conversion is stored in `out[1]`.

float(*x*)

Convert a number, array, or string to a `FloatingPoint` data type. For numeric data, the smallest suitable `FloatingPoint` type is used. Converts strings to `Float64`.

This function is not recommended for arrays. It is better to use a more specific function such as `float32` or `float64`.

significand(*x*)

Extract the significand(s) (a.k.a. mantissa), in binary representation, of a floating-point number or array.

```
julia> significand(15.2)/15.2
0.125
```

```
julia> significand(15.2)*8
15.2
```

exponent(*x*) → `Int`

Get the exponent of a normalized floating-point number.

complex64(*r*[, *i*])

Convert to `r + i*im` represented as a `Complex64` data type. *i* defaults to zero.

complex128(*r*[, *i*])

Convert to `r + i*im` represented as a `Complex128` data type. *i* defaults to zero.

complex(*r*[, *i*])

Convert real numbers or arrays to complex. *i* defaults to zero.

char(*x*)

Convert a number or array to `Char` data type

bswap(*n*)

Byte-swap an integer

num2hex(*f*)

Get a hexadecimal string of the binary representation of a floating point number

hex2num (*str*)

Convert a hexadecimal string to the floating point number it represents

hex2bytes (*s::ASCIIString*)Convert an arbitrarily long hexadecimal string to its binary representation. Returns an `Array{UInt8, 1}`, i.e. an array of bytes.**bytes2hex** (*bin_arr::Array{UInt8, 1}*)Convert an array of bytes to its hexadecimal representation. All characters are in lower-case. Returns an `ASCIIString`.

2.4.3 Numbers

one (*x*)Get the multiplicative identity element for the type of *x* (*x* can also specify the type itself). For matrices, returns an identity matrix of the appropriate size and type.**zero** (*x*)Get the additive identity element for the type of *x* (*x* can also specify the type itself).**pi** π

The constant pi

im

The imaginary unit

e

The constant e

catalan

Catalan's constant

 γ

Euler's constant

 ϕ

The golden ratio

InfPositive infinity of type `Float64`**Inf32**Positive infinity of type `Float32`**Inf16**Positive infinity of type `Float16`**NaN**A not-a-number value of type `Float64`**NaN32**A not-a-number value of type `Float32`**NaN16**A not-a-number value of type `Float16`**issubnormal** (*f*) \rightarrow `Bool`

Test whether a floating point number is subnormal

isfinite (*f*) \rightarrow `Bool`

Test whether a number is finite

isinf (*f*) → Bool

Test whether a number is infinite

isnan (*f*) → Bool

Test whether a floating point number is not a number (NaN)

inf (*f*)

Returns positive infinity of the floating point type *f* or of the same floating point type as *f*

nan (*f*)

Returns NaN (not-a-number) of the floating point type *f* or of the same floating point type as *f*

nextfloat (*f*)

Get the next floating point number in lexicographic order

prevfloat (*f*) → FloatingPoint

Get the previous floating point number in lexicographic order

isinteger (*x*) → Bool

Test whether *x* or all its elements are numerically equal to some integer

isreal (*x*) → Bool

Test whether *x* or all its elements are numerically equal to some real number

BigInt (*x*)

Create an arbitrary precision integer. *x* may be an Int (or anything that can be converted to an Int) or a String. The usual mathematical operators are defined for this type, and results are promoted to a BigInt.

BigFloat (*x*)

Create an arbitrary precision floating point number. *x* may be an Integer, a Float64, a String or a BigInt. The usual mathematical operators are defined for this type, and results are promoted to a BigFloat. Note that because floating-point numbers are not exactly-representable in decimal notation, BigFloat(2.1) may not yield what you expect. You may prefer to initialize constants using strings, e.g., BigFloat("2.1").

get_rounding (*T*)

Get the current floating point rounding mode for type *T*. Valid modes are RoundNearest, RoundToZero, RoundUp, RoundDown, and RoundFromZero (BigFloat only).

set_rounding (*T, mode*)

Set the rounding mode of floating point type *T*. Note that this may affect other types, for instance changing the rounding mode of Float64 will change the rounding mode of Float32. See get_rounding for available modes

with_rounding (*f::Function, T, mode*)

Change the rounding mode of floating point type *T* for the duration of *f*. It is logically equivalent to:

```
old = get_rounding(T)
set_rounding(T, mode)
f()
set_rounding(T, old)
```

See get_rounding for available rounding modes.

Integers

count_ones (*x::Integer*) → Integer

Number of ones in the binary representation of *x*.

```
julia> count_ones(7)
3
```

count_zeros ($x::Integer$) \rightarrow Integer

Number of zeros in the binary representation of x .

```
julia> count_zeros(int32(2 ^ 16 - 1))
16
```

leading_zeros ($x::Integer$) \rightarrow Integer

Number of zeros leading the binary representation of x .

```
julia> leading_zeros(int32(1))
31
```

leading_ones ($x::Integer$) \rightarrow Integer

Number of ones leading the binary representation of x .

```
julia> leading_ones(int32(2 ^ 32 - 2))
31
```

trailing_zeros ($x::Integer$) \rightarrow Integer

Number of zeros trailing the binary representation of x .

```
julia> trailing_zeros(2)
1
```

trailing_ones ($x::Integer$) \rightarrow Integer

Number of ones trailing the binary representation of x .

```
julia> trailing_ones(3)
2
```

isprime ($x::Integer$) \rightarrow Bool

Returns `true` if x is prime, and `false` otherwise.

```
julia> isprime(3)
true
```

primes (n)

Returns a collection of the prime numbers $\leq n$.

isodd ($x::Integer$) \rightarrow Bool

Returns `true` if x is odd (that is, not divisible by 2), and `false` otherwise.

```
julia> isodd(9)
true
```

```
julia> isodd(10)
false
```

iseven ($x::Integer$) \rightarrow Bool

Returns `true` if x is even (that is, divisible by 2), and `false` otherwise.

```
julia> iseven(10)
true
```

```
julia> iseven(9)
false
```

2.4.4 BigFloats

The *BigFloat* type implements arbitrary-precision floating-point arithmetic using the [GNU MPFR library](#).

precision (*num::FloatingPoint*)

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

get_bigfloat_precision ()

Get the precision (in bits) currently used for BigFloat arithmetic.

set_bigfloat_precision (*x::Int64*)

Set the precision (in bits) to be used to BigFloat arithmetic.

with_bigfloat_precision (*f::Function, precision::Integer*)

Change the BigFloat arithmetic precision (in bits) for the duration of *f*. It is logically equivalent to:

```
old = get_bigfloat_precision()
set_bigfloat_precision(precision)
f()
set_bigfloat_precision(old)
```

2.4.5 Random Numbers

Random number generation in Julia uses the [Mersenne Twister library](#). Julia has a global RNG, which is used by default. Multiple RNGs can be plugged in using the `AbstractRNG` object, which can then be used to have multiple streams of random numbers. Currently, only `MersenneTwister` is supported.

rand (*[rng]*, *seed*)

Seed the RNG with a *seed*, which may be an unsigned integer or a vector of unsigned integers. *seed* can even be a filename, in which case the seed is read from a file. If the argument *rng* is not provided, the default global RNG is seeded.

MersenneTwister (*[seed]*)

Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

rand () → `Float64`

Generate a `Float64` random number uniformly in [0,1)

rand! (*[rng]*, *A*)

Populate the array *A* with random number generated from the specified RNG.

rand (*rng::AbstractRNG* [, *dims...*])

Generate a random `Float64` number or array of the size specified by *dims*, using the specified RNG object. Currently, `MersenneTwister` is the only available Random Number Generator (RNG), which may be seeded using `rand`.

rand (*dims* or [*dims...*])

Generate a random `Float64` array of the size specified by *dims*

rand (*Int32|Uint32|Int64|Uint64|Int128|Uint128* [, *dims...*])

Generate a random integer of the given type. Optionally, generate an array of random integers of the given type by specifying *dims*.

rand (*r* [, *dims...*])

Generate a random integer in the range *r* (for example, `1:n` or `0:2:10`). Optionally, generate a random integer array.

randbool (*[dims...]*)

Generate a random boolean value. Optionally, generate an array of random boolean values.

randbool! (*A*)

Fill an array with random boolean values. *A* may be an `Array` or a `BitArray`.

randn (*[rng]*, *dims* or *[dims...]*)

Generate a normally-distributed random number with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers.

randn! (*[rng]*, *A::Array{Float64, N}*)

Fill the array *A* with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the `rand` function.

2.5 Strings

length (*s*)

The number of characters in string *s*.

sizeof (*s::String*)

The number of bytes in string *s*.

***** (*s*, *t*)

Concatenate strings. The `*` operator is an alias to this function.

```
julia> "Hello " * "world"
"Hello world"
```

^ (*s*, *n*)

Repeat *n* times the string *s*. The `^` operator is an alias to this function.

```
julia> "Test " ^ 3
"Test Test Test "
```

string (*xs...*)

Create a string from any values using the `print` function.

repr (*x*)

Create a string from any value using the `showall` function.

bytestring (*::Ptr{UInt8}* [*, length*])

Create a string from the address of a C (0-terminated) string encoded in ASCII or UTF-8. A copy is made; the `ptr` can be safely freed. If `length` is specified, the string does not have to be 0-terminated.

bytestring (*s*)

Convert a string to a contiguous byte array representation appropriate for passing it to C functions. The string will be encoded as either ASCII or UTF-8.

ascii (*::Array{UInt8, I}*)

Create an ASCII string from a byte array.

ascii (*s*)

Convert a string to a contiguous ASCII string (all characters must be valid ASCII characters).

utf8 (*::Array{UInt8, I}*)

Create a UTF-8 string from a byte array.

utf8 (*s*)

Convert a string to a contiguous UTF-8 string (all characters must be valid UTF-8 characters).

normalize_string (*s*, *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 obtained by calling *normalize_string(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)
- stripccc=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*.

is_valid_ascii(*s*) → Bool

Returns true if the string or byte vector is valid ASCII, false otherwise.

is_valid_utf8(*s*) → Bool

Returns true if the string or byte vector is valid UTF-8, false otherwise.

is_valid_char(*c*) → Bool

Returns true if the given char or integer is a valid Unicode code point.

is_assigned_char(*c*) → Bool

Returns true if the given char or integer is an assigned Unicode code point.

ismatch(*r::Regex*, *s::String*) → Bool

Test whether a string contains a match of the given regular expression.

match(*r::Regex*, *s::String*[, *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.

eachmatch(*r::Regex*, *s::String*[, *overlap::Bool=false*])

Search for all matches of a the regular expression *r* in *s* and return an 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.

matchall(*r::Regex*, *s::String*[, *overlap::Bool=false*]) → Vector{String}

Return a vector of the matching substrings from *eachmatch*.

lpad (*string*, *n*, *p*)

Make a string at least *n* characters long by padding on the left with copies of *p*.

rpadd (*string*, *n*, *p*)

Make a string at least *n* characters long by padding on the right with copies of *p*.

search (*string*, *chars*[, *start*])

Search for the first occurrence of the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings). The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that `s[search(s, x)] == x`:

```
search(string, "substring") = start:end such that string[start:end] ==
"substring", or 0:-1 if unmatched.
```

```
search(string, 'c') = index such that string[index] == 'c', or 0 if unmatched.
```

rsearch (*string*, *chars*[, *start*])

Similar to `search`, but returning the last occurrence of the given characters within the given string, searching in reverse from *start*.

searchindex (*string*, *substring*[, *start*])

Similar to `search`, but return only the start index at which the substring is found, or 0 if it is not.

rsearchindex (*string*, *substring*[, *start*])

Similar to `rsearch`, but return only the start index at which the substring is found, or 0 if it is not.

contains (*haystack*, *needle*)

Determine whether the second argument is a substring of the first.

replace (*string*, *pat*, *r*[, *n*])

Search for the given pattern *pat*, and replace each occurrence with *r*. If *n* is provided, replace at most *n* occurrences. As with `search`, the second argument 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.

split (*string*, [*chars*, [*limit*,] [*include_empty*]])

Return an array of substrings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by `search`'s second argument (i.e. a single character, collection of characters, string, or regular expression). If *chars* is omitted, it defaults to the set of all space characters, and *include_empty* is taken to be false. The last two arguments are also optional: they are a maximum size for the result and a flag determining whether empty fields should be included in the result.

rsplit (*string*, [*chars*, [*limit*,] [*include_empty*]])

Similar to `split`, but starting from the end of the string.

strip (*string*[, *chars*])

Return *string* with any leading and trailing whitespace removed. If *chars* (a character, or vector or set of characters) is provided, instead remove characters contained in it.

lstrip (*string*[, *chars*])

Return *string* with any leading whitespace removed. If *chars* (a character, or vector or set of characters) is provided, instead remove characters contained in it.

rstrip (*string*[, *chars*])

Return *string* with any trailing whitespace removed. If *chars* (a character, or vector or set of characters) is provided, instead remove characters contained in it.

startswith (*string*, *prefix* | *chars*)

Returns true if *string* starts with *prefix*. If the second argument is a vector or set of characters, tests

whether the first character of `string` belongs to that set.

endswith (*string*, *suffix* | *chars*)

Returns `true` if `string` ends with `suffix`. If the second argument is a vector or set of characters, tests whether the last character of `string` belongs to that set.

uppercase (*string*)

Returns `string` with all characters converted to uppercase.

lowercase (*string*)

Returns `string` with all characters converted to lowercase.

ucfirst (*string*)

Returns `string` with the first character converted to uppercase.

lcfirst (*string*)

Returns `string` with the first character converted to lowercase.

join (*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. For example, `join(["apples", "bananas", "pineapples"], ", ", " and ") == "apples, bananas and pineapples"`.

`strings` can be any iterable over elements `x` which are convertible to strings via `print(io::IOBuffer, x)`.

chop (*string*)

Remove the last character from a string

chomp (*string*)

Remove a trailing newline from a string

ind2chr (*string*, *i*)

Convert a byte index to a character index

chr2ind (*string*, *i*)

Convert a character index to a byte index

isvalid (*str*, *i*)

Tells whether index `i` is valid for the given string

nextind (*str*, *i*)

Get the next valid string index after `i`. Returns a value greater than `endof(str)` at or after the end of the string.

prevind (*str*, *i*)

Get the previous valid string index before `i`. Returns a value less than 1 at the beginning of the string.

randstring (*len*)

Create a random ASCII string of length `len`, consisting of upper- and lower-case letters and the digits 0-9

charwidth (*c*)

Gives the number of columns needed to print a character.

strwidth (*s*)

Gives the number of columns needed to print a string.

isalnum (*c::Union{Char, String}*) → Bool

Tests whether a character is alphanumeric, or whether this is true for all elements of a string.

isalpha (*c::Union{Char, String}*) → Bool

Tests whether a character is alphabetic, or whether this is true for all elements of a string.

isascii (*c::Union{Char, String}*) → Bool

Tests whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

isblank (*c::Union{Char, String}*) → Bool

Tests whether a character is a tab or space, or whether this is true for all elements of a string.

iscntrl (*c::Union{Char, String}*) → Bool

Tests whether a character is a control character, or whether this is true for all elements of a string.

isdigit (*c::Union{Char, String}*) → Bool

Tests whether a character is a numeric digit (0-9), or whether this is true for all elements of a string.

isgraph (*c::Union{Char, String}*) → Bool

Tests whether a character is printable, and not a space, or whether this is true for all elements of a string.

islower (*c::Union{Char, String}*) → Bool

Tests whether a character is a lowercase letter, or whether this is true for all elements of a string.

isprint (*c::Union{Char, String}*) → Bool

Tests whether a character is printable, including space, or whether this is true for all elements of a string.

ispunct (*c::Union{Char, String}*) → Bool

Tests whether a character is printable, and not a space or alphanumeric, or whether this is true for all elements of a string.

isspace (*c::Union{Char, String}*) → Bool

Tests whether a character is any whitespace character, or whether this is true for all elements of a string.

isupper (*c::Union{Char, String}*) → Bool

Tests whether a character is an uppercase letter, or whether this is true for all elements of a string.

isxdigit (*c::Union{Char, String}*) → Bool

Tests whether a character is a valid hexadecimal digit, or whether this is true for all elements of a string.

symbol (*str*) → Symbol

Convert a string to a Symbol.

escape_string (*str::String*) → String

General escaping of traditional C and Unicode escape sequences. See `print_escaped()` for more general escaping.

unescape_string (*s::String*) → String

General unescaping of traditional C and Unicode escape sequences. Reverse of `escape_string()`. See also `print_unescaped()`.

utf16 (*s*)

Create a UTF-16 string from a byte array, array of `UInt16`, or any other string type. (Data must be valid UTF-16. Conversions of byte arrays check for a byte-order marker in the first two bytes, and do not include it in the resulting string.)

Note that the resulting `UTF16String` data is terminated by the NUL codepoint (16-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the `utf16(s)` conversion function. If you have a `UInt16` array *A* that is already NUL-terminated valid UTF-16 data, then you can instead use `UTF16String(A)` to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

utf16 (*::Union{Ptr{UInt16}, Ptr{Int16}}*) [*length*]

Create a string from the address of a NUL-terminated UTF-16 string. A copy is made; the pointer can be safely freed. If *length* is specified, the string does not have to be NUL-terminated.

is_valid_utf16(*s*) → Bool

Returns true if the string or `UInt16` array is valid UTF-16.

utf32(*s*)

Create a UTF-32 string from a byte array, array of `UInt32`, or any other string type. (Conversions of byte arrays check for a byte-order marker in the first four bytes, and do not include it in the resulting string.)

Note that the resulting `UTF32String` data is terminated by the NUL codepoint (32-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the `utf32(s)` conversion function. If you have a `UInt32` array *A* that is already NUL-terminated UTF-32 data, then you can instead use `UTF32String(A)` to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

utf32(*::Union{Ptr{Char}, Ptr{UInt32}, Ptr{Int32}}*[], *length*)

Create a string from the address of a NUL-terminated UTF-32 string. A copy is made; the pointer can be safely freed. If *length* is specified, the string does not have to be NUL-terminated.

wstring(*s*)

This is a synonym for either `utf32(s)` or `utf16(s)`, depending on whether `Cwchar_t` is 32 or 16 bits, respectively. The synonym `WString` for `UTF32String` or `UTF16String` is also provided.

2.6 Arrays

2.6.1 Basic functions

ndims(*A*) → Integer

Returns the number of dimensions of *A*

size(*A*)

Returns a tuple containing the dimensions of *A*

iseltype(*A*, *T*)

Tests whether *A* or its elements are of type *T*

length(*A*) → Integer

Returns the number of elements in *A*

countnz(*A*)

Counts the number of nonzero values in array *A* (dense or sparse). Note that this is not a constant-time operation. For sparse matrices, one should usually use `nnz`, which returns the number of stored values.

conj!(*A*)

Convert an array to its complex conjugate in-place

stride(*A*, *k*)

Returns the distance in memory (in number of elements) between adjacent elements in dimension *k*

strides(*A*)

Returns a tuple of the memory strides in each dimension

ind2sub(*dims*, *index*) → subscripts

Returns a tuple of subscripts into an array with dimensions *dims*, corresponding to the linear index *index*

Example `i, j, ... = ind2sub(size(A), indmax(A))` provides the indices of the maximum element

sub2ind(*dims*, *i*, *j*, *k*...) → index

The inverse of `ind2sub`, returns the linear index corresponding to the provided subscripts

2.6.2 Constructors

Array (*type, dims*)

Construct an uninitialized dense array. *dims* may be a tuple or a series of integer arguments.

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, ...]`.

cell (*dims*)

Construct an uninitialized cell array (heterogeneous array). *dims* can be either a tuple or a series of integer arguments.

zeros (*type, dims*)

Create an array of all zeros of specified type. The type defaults to `Float64` if not specified.

zeros (*A*)

Create an array of all zeros with the same element type and shape as *A*.

ones (*type, dims*)

Create an array of all ones of specified type. The type defaults to `Float64` if not specified.

ones (*A*)

Create an array of all ones with the same element type and shape as *A*.

trues (*dims*)

Create a `BitArray` with all values set to `true`

falses (*dims*)

Create a `BitArray` with all values set to `false`

fill (*x, dims*)

Create an array filled with the value *x*. For example, `fill(1.0, (10, 10))` returns a 10x10 array of floats, with each element initialized to 1.0.

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.

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.

reshape (*A, dims*)

Create an array with the same data as the given array, but with different dimensions. An implementation for a particular type of array may choose whether the data is copied or shared.

similar (*array, element_type, dims*)

Create an uninitialized array of the same type as the given array, but with the specified element type and dimensions. The second and third arguments are both optional. The *dims* argument may be a tuple or a series of integer arguments.

reinterpret (*type, A*)

Change the type-interpretation of a block of memory. For example, `reinterpret(Float32, uint32(7))` interprets the 4 bytes corresponding to `uint32(7)` as a `Float32`. For arrays, this constructs an array with the same binary data as the given array, but with the specified element type.

eye (*n*)

n-by-*n* identity matrix

eye (*m, n*)

m-by-*n* identity matrix

eye (*A*)

Constructs an identity matrix of the same dimensions and type as *A*.

linspace (*start, stop, n*)

Construct a vector of *n* linearly-spaced elements from *start* to *stop*. See also: `linrange()` that constructs a range object.

logspace (*start, stop, n*)

Construct a vector of *n* logarithmically-spaced numbers from 10^{start} to 10^{stop} .

2.6.3 Mathematical operators and functions

All mathematical operations and functions are supported for arrays

broadcast (*f, As...*)

Broadcasts the arrays *As* to a common size by expanding singleton dimensions, and returns an array of the results `f(as...)` for each position.

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)`.

bitbroadcast (*f, As...*)

Like `broadcast`, but allocates a `BitArray` to store the result, rather than an `Array`.

broadcast_function (*f*)

Returns a function `broadcast_f` such that `broadcast_function(f)(As...)` `=== broadcast(f, As...)`. Most useful in the form `const broadcast_f = broadcast_function(f)`.

broadcast!_function (*f*)

Like `broadcast_function`, but for `broadcast!`.

2.6.4 Indexing, Assignment, and Concatenation

getindex (*A, inds...*)

Returns a subset of array *A* as specified by *inds*, where each *ind* may be an `Int`, a `Range`, or a `Vector`.

sub (*A, inds...*)

Returns a `SubArray`, which stores the input *A* and *inds* rather than computing the result immediately. Calling `getindex` on a `SubArray` computes the indices on the fly.

parent (*A*)

Returns the “parent array” of an array view type (e.g., `SubArray`), or the array itself if it is not a view

parentindexes (*A*)

From an array view *A*, returns the corresponding indexes in the parent

slicedim (*A, d, i*)

Return all the data of *A* where the index for dimension *d* equals *i*. Equivalent to `A[:, :, ..., i, :, :, ...]` where *i* is in position *d*.

slice (*A, inds...*)

Create a view of the given indexes of array *A*, dropping dimensions indexed with scalars.

setindex! (*A, X, inds...*)

Store values from array *X* within some subset of *A* as specified by *inds*.

broadcast_getindex (*A*, *inds...*)

Broadcasts the *inds* arrays to a common size like `broadcast`, and returns an array of the results `A[ks...]`, where *ks* goes over the positions in the broadcast.

broadcast_setindex! (*A*, *X*, *inds...*)

Broadcasts the *X* and *inds* arrays to a common size and stores the value from each position in *X* at the indices given by the same positions in *inds*.

cat (*dim*, *A...*)

Concatenate the input arrays along the specified dimension

vcat (*A...*)

Concatenate along dimension 1

hcat (*A...*)

Concatenate along dimension 2

hvcat (*rows::(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. For example, `[a b; c d e]` calls `hvcat((2, 3), a, b, c, d, e)`.

If the first argument is a single integer *n*, then all block rows are assumed to have *n* block columns.

flipdim (*A*, *d*)

Reverse *A* in dimension *d*.

flipud (*A*)

Equivalent to `flipdim(A, 1)`.

fliplr (*A*)

Equivalent to `flipdim(A, 2)`.

circshift (*A*, *shifts*)

Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

find (*A*)

Return a vector of the linear indexes of the non-zeros in *A* (determined by `A[i] != 0`). A common use of this is to convert a boolean array to an array of indexes of the `true` elements.

find (*f*, *A*)

Return a vector of the linear indexes of *A* where *f* returns true.

findn (*A*)

Return a vector of indexes for each dimension giving the locations of the non-zeros in *A* (determined by `A[i] != 0`).

findnz (*A*)

Return a tuple (*I*, *J*, *V*) where *I* and *J* are the row and column indexes of the non-zero values in matrix *A*, and *V* is a vector of the non-zero values.

findfirst (*A*)

Return the index of the first non-zero value in *A* (determined by `A[i] != 0`).

findfirst (*A*, *v*)

Return the index of the first element equal to *v* in *A*.

findfirst (*predicate*, *A*)

Return the index of the first element of *A* for which *predicate* returns true.

findnext (*A*, *i*)

Find the next index $\geq i$ of a non-zero element of *A*, or 0 if not found.

findnext (*predicate*, *A*, *i*)

Find the next index $\geq i$ of an element of *A* for which *predicate* returns true, or 0 if not found.

findnext (*A*, *v*, *i*)

Find the next index $\geq i$ of an element of *A* equal to *v* (using `==`), or 0 if not found.

permutedims (*A*, *perm*)

Permute the dimensions of array *A*. *perm* is a vector specifying a permutation of length `ndims(A)`. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to `permutedims(A, [2, 1])`.

ipermutedims (*A*, *perm*)

Like `permutedims()`, except the inverse of the given permutation is applied.

squeeze (*A*, *dims*)

Remove the dimensions specified by *dims* from array *A*

vec (*Array*) \rightarrow Vector

Vectorize an array using column-major convention.

promote_shape (*s1*, *s2*)

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

checkbounds (*array*, *indexes...*)

Throw an error if the specified indexes are not in bounds for the given array.

randsubseq (*A*, *p*) \rightarrow 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*.

randsubseq! (*S*, *A*, *p*)

Like `randsubseq`, but the results are stored in *S* (which is resized as needed).

2.6.5 Array functions

cumprod (*A*[:, *dim*])

Cumulative product along a dimension.

cumprod! (*B*, *A*[:, *dim*])

Cumulative product of *A* along a dimension, storing the result in *B*.

cumsum (*A*[:, *dim*])

Cumulative sum along a dimension.

cumsum! (*B*, *A*[:, *dim*])

Cumulative sum of *A* along a dimension, storing the result in *B*.

cumsum_kbn (*A*[:, *dim*])

Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

cummin (*A*[:, *dim*])

Cumulative minimum along a dimension.

cummax (*A*[:, *dim*])

Cumulative maximum along a dimension.

diff (*A*[:, *dim*])

Finite difference operator of matrix or vector.

gradient ($F[h]$)

Compute differences along vector F , using h as the spacing between points. The default spacing is one.

rot180 (A)

Rotate matrix A 180 degrees.

rot180 (A, k)

Rotate matrix A 180 degrees an integer k number of times. If k is even, this is equivalent to a `copy`.

rotl90 (A)

Rotate matrix A left 90 degrees.

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`.

rotr90 (A)

Rotate matrix A right 90 degrees.

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`.

reducedim ($f, A, dims, initial$)

Reduce 2-argument function f along dimensions of A . $dims$ is a vector specifying the dimensions to reduce, and $initial$ is the initial value to use in the reductions.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for `reduce`.

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 $A[\dots, :, \dots, :, \dots]$. $dims$ 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 .

sum_kbn (A)

Returns the sum of all array elements, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

cartesianmap ($f, dims$)

Given a $dims$ tuple of integers (m, n, \dots) , call f on all combinations of integers in the ranges $1:m, 1:n$, etc.

```
julia> cartesianmap(println, (2,2))
11
21
12
22
```

2.6.6 Combinatorics

nthperm (v, k)

Compute the k th lexicographic permutation of a vector.

nthperm (p)

Return the k that generated permutation p . Note that `nthperm(nthperm([1:n], k)) == k` for $1 \leq k \leq \text{factorial}(n)$.

nthperm! (*v*, *k*)

In-place version of `nthperm()`.

randperm (*n*)

Construct a random permutation of the given length.

invperm (*v*)

Return the inverse permutation of *v*.

isperm (*v*) → Bool

Returns true if *v* is a valid permutation.

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.

ipermute! (*v*, *p*)

Like `permute!`, but the inverse of the given permutation is applied.

randcycle (*n*)

Construct a random cyclic permutation of the given length.

shuffle (*v*)

Return a randomly permuted copy of *v*.

shuffle! (*v*)

In-place version of `shuffle()`.

reverse (*v*[, *start*=1[, *stop*=length(*v*)]])

Return a copy of *v* reversed from *start* to *stop*.

reverse! (*v*[, *start*=1[, *stop*=length(*v*)]]) → *v*

In-place version of `reverse()`.

combinations (*arr*, *n*)

Generate all combinations of *n* elements from an indexable object. Because the number of combinations can be very large, this function returns an iterator object. Use `collect(combinations(a, n))` to get an array of all combinations.

permutations (*arr*)

Generate all permutations of an indexable object. Because the number of permutations can be very large, this function returns an iterator object. Use `collect(permutations(a, n))` to get an array of all permutations.

partitions (*n*)

Generate all integer arrays that sum to *n*. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(n))`.

partitions (*n*, *m*)

Generate all arrays of *m* integers that sum to *n*. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n, m))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(n, m))`.

partitions (*array*)

Generate all set partitions of the elements of an array, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array))` to get an array of all partitions. The number of partitions to generate can be efficiently computed using `length(partitions(array))`.

partitions (*array*, *m*)

Generate all set partitions of the elements of an array into exactly *m* subsets, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array,m))` to get an array of all partitions. The number of partitions into *m* subsets is equal to the Stirling number of the second kind and can be efficiently computed using `length(partitions(array,m))`.

2.6.7 BitArrays

bitpack (*A::AbstractArray{T,N}*) → BitArray

Converts a numeric array to a packed boolean array

bitunpack (*B::BitArray{N}*) → Array{Bool,N}

Converts a packed boolean array to an array of booleans

flipbits! (*B::BitArray{N}*) → BitArray{N}

Performs a bitwise not operation on *B*. See `~ operator`.

rol (*B::BitArray{1}*, *i::Integer*) → BitArray{1}

Left rotation operator.

ror (*B::BitArray{1}*, *i::Integer*) → BitArray{1}

Right rotation operator.

2.6.8 Sparse Matrices

Sparse matrices support much of the same set of operations as dense matrices. The following functions are specific to sparse matrices.

sparse (*I*, *J*, *V* [, *m*, *n*, *combine*])

Create a sparse matrix *S* of dimensions *m* × *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 `max(I)` and `max(J)` respectively. If the `combine` function is not supplied, duplicates are added by default.

sparsevec (*I*, *V* [, *m*, *combine*])

Create a sparse matrix *S* of size *m* × 1 such that *S*[*I*[*k*]] = *V*[*k*]. Duplicates are combined using the `combine` function, which defaults to + if it is not provided. In julia, sparse vectors are really just sparse matrices with one column. Given Julia's Compressed Sparse Columns (CSC) storage format, a sparse column matrix with one column is sparse, whereas a sparse row matrix with one row ends up being dense.

sparsevec (*D::Dict{,m}*)

Create a sparse matrix of size *m* × 1 where the row values are keys from the dictionary, and the nonzero values are the values from the dictionary.

issparse (*S*)

Returns `true` if *S* is sparse, and `false` otherwise.

sparse (*A*)

Convert a dense matrix *A* into a sparse matrix.

sparsevec (*A*)

Convert a dense vector *A* into a sparse matrix of size *m* × 1. In julia, sparse vectors are really just sparse matrices with one column.

full (*S*)

Convert a sparse matrix *S* into a dense matrix.

nnz (*A*)

Returns the number of stored (filled) elements in a sparse matrix.

spzeros (*m*, *n*)

Create an empty sparse matrix of size $m \times n$.

spones (*S*)

Create a sparse matrix with the same structure as that of *S*, but with every nonzero element having the value 1.0.

speye (*type*, *m* [, *n*])

Create a sparse identity matrix of specified type of size $m \times m$. In case *n* is supplied, create a sparse identity matrix of size $m \times n$.

spdiags (*B*, *d* [, *m*, *n*])

Construct a sparse diagonal matrix. *B* is a tuple of vectors containing the diagonals and *d* is a tuple containing the positions of the diagonals. In the case the input contains only one diagonal, *B* can be a vector (instead of a tuple) and *d* can be the diagonal position (instead of a tuple), defaulting to 0 (diagonal). Optionally, *m* and *n* specify the size of the resulting sparse matrix.

sprand (*m*, *n*, *p* [, *rng*])

Create a random 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 *rng*. The uniform distribution is used in case *rng* is not specified.

sprandn (*m*, *n*, *p*)

Create a random m by n sparse matrix with the specified (independent) probability *p* of any entry being nonzero, where nonzero values are sampled from the normal distribution.

sprandbool (*m*, *n*, *p*)

Create a random m by n sparse boolean matrix with the specified (independent) probability *p* of any entry being true.

etree (*A* [, *post*])

Compute the elimination tree of a symmetric sparse matrix *A* from `triu(A)` and, optionally, its post-ordering permutation.

symperm (*A*, *p*)

Return the symmetric permutation of *A*, which is $A[p, p]$. *A* should be symmetric and sparse, where only the upper triangular part of the matrix is stored. This algorithm ignores the lower triangular part of the matrix. Only the upper triangular part of the result is returned as well.

nonzeros (*A*)

Return a vector of the structural nonzero values in sparse matrix *A*. This includes zeros that are explicitly stored in the sparse matrix. The returned vector points directly to the internal nonzero storage of *A*, and any modifications to the returned vector will mutate *A* as well.

2.7 Tasks and Parallel Computing

2.7.1 Tasks

Task (*func*)

Create a Task (i.e. thread, or coroutine) to execute the given function (which must be callable with no arguments). The task exits when this function returns.

yieldto (*task*, *args*...)

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, `args` are 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.

current_task()

Get the currently running Task.

istaskdone(*task*) → Bool

Tell whether a task has exited.

consume(*task, values...*)

Receive the next value passed to `produce` by the specified task. Additional arguments may be passed, to be returned from the last `produce` call in the producer.

produce(*value*)

Send the given value to the last `consume` call, switching to the consumer task. If the next `consume` call passes any values, they are returned by `produce`.

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.

task_local_storage(*symbol*)

Look up the value of a symbol in the current task's task-local storage.

task_local_storage(*symbol, value*)

Assign a value to a symbol in the current task's task-local storage.

task_local_storage(*body, symbol, value*)

Call the function `body` with a modified task-local storage, in which `value` is assigned to `symbol`; the previous value of `symbol`, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

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 `RemoteRef` type does this, and so can be used for level-triggered events.

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.

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 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.

@schedule()

Wrap an expression in a Task and add it to the scheduler's queue.

@task()

Wrap an expression in a Task executing it, and return the Task. This only creates a task, and does not run it.

sleep(*seconds*)

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of 0.001.

2.7.2 General Parallel Computing Support

addprocs (*n*; *cman*::ClusterManager=LocalManager()) → List of process identifiers

`addprocs(4)` will add 4 processes on the local machine. This can be used to take advantage of multiple cores.

Keyword argument `cman` can be used to provide a custom cluster manager to start workers. For example Beowulf clusters are supported via a custom cluster manager implemented in package `ClusterManagers`.

See the documentation for package `ClusterManagers` for more information on how to write a custom cluster manager.

addprocs (*machines*; *tunnel*=false, *dir*=JULIA_HOME, *sshflags*::Cmd="" → 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 host definitions of the form `[user@]host[:port] [bind_addr]`. `user` defaults to current user, `port` to the standard ssh port. Optionally, in case of multi-homed hosts, `bind_addr` may be used to explicitly specify an interface.

Keyword arguments:

`tunnel` : if `true` then SSH tunneling will be used to connect to the worker.

`dir` : specifies the location of the `julia` binaries on the worker nodes.

`sshflags` : specifies additional ssh options, e.g. `sshflags="-i /home/foo/bar.pem"`.

nprocs ()

Get the number of available processes.

nworkers ()

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

procs ()

Returns a list of all process identifiers.

workers ()

Returns a list of all worker process identifiers.

rmprocs (*pids*...)

Removes the specified workers.

interrupt ([*pids*...])

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.

myid ()

Get the id of the current process.

pmap (*f*, *lists*...; *err_retry*=true, *err_stop*=false)

Transform collections `lists` by applying `f` to each element in parallel. If `nprocs() > 1`, the calling process will be dedicated to assigning tasks. All other available processes will be used as parallel workers.

If `err_retry` is true, it retries a failed application of `f` on a different worker. If `err_stop` is true, it takes precedence over the value of `err_retry` and `pmap` stops execution on the first error.

remotecall (*id*, *func*, *args*...)

Call a function asynchronously on the given arguments on the specified process. Returns a `RemoteRef`.

wait ([*x*])

Block the current task until some event occurs, depending on the type of the argument:

- `RemoteRef`: Wait for a value to become available for the specified remote reference.

- 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, returning its result value.
- RawFD**: Wait for changes on a file descriptor (see `poll_fd` for keyword arguments and return code)

If no argument is passed, the task blocks for an undefined period. If the task's state is set to `:waiting`, it can only be restarted by an explicit call to `schedule` or `yieldto`. If the task's state is `:runnable`, it might be restarted unpredictably.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

fetch (*RemoteRef*)

Wait for and get the value of a remote reference.

remotecall_wait (*id, func, args...*)

Perform `wait(remotecall(...))` in one message.

remotecall_fetch (*id, func, args...*)

Perform `fetch(remotecall(...))` in one message.

put! (*RemoteRef, value*)

Store a value to a remote reference. Implements “shared queue of length 1” semantics: if a value is already present, blocks until the value is removed with `take!`. Returns its first argument.

take! (*RemoteRef*)

Fetch the value of a remote reference, removing it so that the reference is empty again.

isready (*r::RemoteRef*)

Determine whether a `RemoteRef` 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. It is recommended that this function only be used on a `RemoteRef` that is assigned once.

If the argument `RemoteRef` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `r` in a separate task instead, or to use a local `RemoteRef` as a proxy:

```
rr = RemoteRef()
@async put!(rr, remotecall_fetch(p, long_computation))
isready(rr) # will not block
```

RemoteRef ()

Make an uninitialized remote reference on the local machine.

RemoteRef (*n*)

Make an uninitialized remote reference on process `n`.

timedwait (*testcb::Function, secs::Float64; pollint::Float64=0.1*)

Waits till `testcb` returns `true` or for `secs` ` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds.

@spawn ()

Execute an expression on an automatically-chosen process, returning a `RemoteRef` to the result.

@spawnat ()

Accepts two arguments, `p` and an expression, and runs the expression asynchronously on process `p`, returning a `RemoteRef` to the result.

@fetch ()

Equivalent to `fetch(@spawn expr)`.

@fetchfrom()

Equivalent to `fetch(@spawnat p expr)`.

@async()

Schedule an expression to run on the local machine, also adding it to the set of items that the nearest enclosing `@sync` waits for.

@sync()

Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete.

@parallel()

A parallel for loop of the form

```
@parallel [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, `@parallel` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@parallel` 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 @parallel for var = range
    body
end
```

2.7.3 Distributed Arrays

DArray (*init*, *dims*[, *procs*, *dist*])

Construct a distributed array. The parameter *init* is a function that accepts a tuple of index ranges. This function should allocate a local chunk of the distributed array and initialize it for the specified indices. *dims* is the overall size of the distributed array. *procs* optionally specifies a vector of process IDs to use. If unspecified, the array is distributed over all worker processes only. Typically, when running in distributed mode, i.e., `nprocs() > 1`, this would mean that no chunk of the distributed array exists on the process hosting the interactive julia prompt. *dist* is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.

For example, the `dfill` function that creates a distributed array and fills it with a value *v* is implemented as:

```
dfill(v, args...) = DArray{I->fill(v, map(length,I)), args...}
```

dzeros (*dims*, ...)

Construct a distributed array of zeros. Trailing arguments are the same as those accepted by `DArray()`.

dones (*dims*, ...)

Construct a distributed array of ones. Trailing arguments are the same as those accepted by `DArray()`.

dfill (*x*, *dims*, ...)

Construct a distributed array filled with value *x*. Trailing arguments are the same as those accepted by `DArray()`.

drand (*dims*, ...)

Construct a distributed uniform random array. Trailing arguments are the same as those accepted by `DArray()`.

drandn (*dims*, ...)

Construct a distributed normal random array. Trailing arguments are the same as those accepted by `DArray()`.

distribute (*a*)

Convert a local array to distributed.

localpart (*d*)

Get the local piece of a distributed array. Returns an empty array if no local part exists on the calling process.

localindexes (*d*)

A tuple describing the indexes owned by the local process. Returns a tuple with empty ranges if no local part exists on the calling process.

procs (*d*)

Get the vector of processes storing pieces of *d*.

2.7.4 Shared Arrays (Experimental, UNIX-only feature)

SharedArray (*T::Type, dims::NTuple; init=false, pids=Int[]*)

Construct a SharedArray of a bitstype *T* and size *dims* across the processes specified by *pids* - all of which have to be on the same host.

If *pids* is left unspecified, the shared array will be mapped across all workers on the current host.

If an *init* function of the type *initfn* (*S::SharedArray*) is specified, it is called on all the participating workers.

procs (*S::SharedArray*)

Get the vector of processes that have mapped the shared array

sdata (*S::SharedArray*)

Returns the actual `Array` object backing *S*

indexpids (*S::SharedArray*)

Returns the index of the current worker into the *pids* vector, i.e., the list of workers mapping the SharedArray

2.8 Linear Algebra

2.8.1 Standard Functions

Linear algebra functions in Julia are largely implemented by calling functions from [LAPACK](#). Sparse factorizations call functions from [SuiteSparse](#).

***** (*A, B*)

Matrix multiplication

**** (*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*. A direct solver is used for upper- or lower triangular *A*. For Hermitian *A* (equivalent to symmetric *A* for non-complex *A*) the `BunchKaufman` factorization is used. Otherwise 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. For sparse, square *A* the LU factorization (from `UMFPACK`) is used.

dot (*x, y*)

\cdot (*x, y*)

Compute the dot product. For complex vectors, the first vector is conjugated.

cross (*x, y*)

x (*x*, *y*)

Compute the cross product of two 3-vectors.

rref (*A*)

Compute the reduced row echelon form of the matrix *A*.

factorize (*A*)

Compute a convenient factorization (including LU, Cholesky, Bunch-Kaufman, Triangular) of *A*, based upon the type of the input matrix. The return value can then be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C`.

factorize! (*A*)

`factorize!` is the same as `factorize()`, but saves space by overwriting the input *A*, instead of creating a copy.

lu (*A*) → *L*, *U*, *p*

Compute the LU factorization of *A*, such that $A[p, :] = L*U$.

lufact (*A*[, *pivot=true*]) → *F*

Compute the LU factorization of *A*. The return type of *F* depends on the type of *A*. In most cases, if *A* is a subtype *S* of `AbstractMatrix` 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 `<`. When *A* is sparse and have element of type `Float32`, `Float64`, `Complex{Float32}`, or `Complex{Float64}` the return type is `UmfpackLU`. Some examples are shown in the table below.

Type of input <i>A</i>	Type of output <i>F</i>	Relationship between <i>F</i> and <i>A</i>
<code>Matrix()</code>	<code>LU</code>	$F[:, L]*F[:, U] == A[F[:, p], :]$
<code>Tridiagonal()</code>	<code>LU{T, Tridiagonal{T}}</code>	$F[:, L]*F[:, U] == A$
<code>SparseMatrixCSC{<i>T</i>}</code>	<code>UmfpackLU</code>	$F[:, L]*F[:, U] == F[:, Rs] .* A[F[:, p], F[:, q]]$

The individual components of the factorization *F* can be accessed by indexing:

Component	Description	LU	<code>LU{T, Tridiagonal{T}}</code>	<code>UmfpackLU</code>
<code>F[:, L]</code>	<i>L</i> (lower triangular) part of <i>LU</i>	x		x
<code>F[:, U]</code>	<i>U</i> (upper triangular) part of <i>LU</i>	x		x
<code>F[:, p]</code>	(right) permutation Vector	x		x
<code>F[:, P]</code>	(right) permutation Matrix	x		
<code>F[:, q]</code>	left permutation Vector			x
<code>F[:, Rs]</code>	Vector of scaling factors			x
<code>F[:, (:)]</code>	(<i>L</i> , <i>U</i> , <i>p</i> , <i>q</i> , <i>Rs</i>) components			x

Supported function	LU	<code>LU{T, Tridiagonal{T}}</code>	<code>UmfpackLU</code>
<code>/</code>	x		
<code>\</code>	x	x	x
<code>cond</code>	x		x
<code>det</code>	x	x	x
<code>size</code>	x	x	

lufact! (*A*) → *LU*

`lufact!` is the same as `lufact()`, but saves space by overwriting the input *A*, instead of creating a copy. For sparse *A* the `nzval` field is not overwritten but the index fields, `colptr` and `rowval` are decremented in place, converting from 1-based indices to 0-based indices.

chol (A , LU) $\rightarrow F$

Compute the Cholesky factorization of a symmetric positive definite matrix A and return the matrix F . If LU is $:L$ (Lower), $A = L \cdot L'$. If LU is $:U$ (Upper), $A = R' \cdot R$.

chofact (A , [LU], [$pivot=false$], [$tol=-1.0$]) \rightarrow Cholesky

Compute the Cholesky factorization of a dense symmetric positive (semi)definite matrix A and return either a Cholesky if $pivot=false$ or CholeskyPivoted if $pivot=true$. LU may be $:L$ for using the lower part or $:U$ for the upper part. The default is to use $:U$. The triangular matrix can be obtained from the factorization F with: $F[:L]$ and $F[:U]$. The following functions are available for Cholesky objects: `size`, `\`, `inv`, `det`. For CholeskyPivoted there is also defined a `rank`. If $pivot=false$ a `PosDefException` exception is thrown in case the matrix is not positive definite. The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

cholfact (A , ll) \rightarrow CholmodFactor

Compute the sparse Cholesky factorization of a sparse matrix A . If A is Hermitian its Cholesky factor is determined. If A is not Hermitian the Cholesky factor of $A \cdot A'$ is determined. A fill-reducing permutation is used. Methods for `size`, `solve`, `\`, `findn_nzs`, `diag`, `det` and `logdet` are available for CholmodFactor objects. One of the solve methods includes an integer argument that can be used to solve systems involving parts of the factorization only. The optional boolean argument, `ll` determines whether the factorization returned is of the $A[p,p] = L \cdot L'$ form, where L is lower triangular or $A[p,p] = L \cdot \text{Diagonal}(D) \cdot L'$ form where L is unit lower triangular and D is a non-negative vector. The default is LDL. The symbolic factorization can also be reused for other matrices with the same structure as A by calling `cholfact!`.

cholfact! (A , [LU], [$pivot=false$], [$tol=-1.0$]) \rightarrow Cholesky

`cholfact!` is the same as `chofact()`, but saves space by overwriting the input A , instead of creating a copy. `cholfact!` can also reuse the symbolic factorization from a different matrix F with the same structure when used as: `cholfact!(F::CholmodFactor, A)`.

ldltfact (A) \rightarrow LDLtFactorization

Compute a factorization of a positive definite matrix A such that $A = L \cdot \text{Diagonal}(d) \cdot L'$ where L is a unit lower triangular matrix and d is a vector with non-negative elements.

qr (A , [$pivot=false$], [$thin=true$]) $\rightarrow Q, R, [p]$

Compute the (pivoted) QR factorization of A such that either $A = Q \cdot R$ or $A[:,p] = Q \cdot R$. Also see `qrfact`. The default is to compute a thin factorization. Note that R is not extended with zeros when the full Q is requested.

qrfact (A , [$pivot=false$]) $\rightarrow F$

Computes the QR factorization of A . The return type of F depends on the element type of A and whether pivoting is specified (with $pivot=true$).

Return type	eltype(A)	pivot	Relationship between F and A
QR	not BlasFloat	either	$A == F[:,Q] \cdot F[:,R]$
QRCompactWY	BlasFloat	false	$A == F[:,Q] \cdot F[:,R]$
QRPivoted	BlasFloat	true	$A[:,F[:,p]] == F[:,Q] \cdot F[:,R]$

BlasFloat refers to any of: Float32, Float64, Complex64 or Complex128.

The individual components of the factorization F can be accessed by indexing:

Component	Description	QR	QRCompactWY	QRPivoted
$F[:,Q]$	Q (orthogonal/unitary) part of QR	x (QRPackedQ)	x (QRCompactWYQ)	x (QRPackedQ)
$F[:,R]$	R (upper right triangular) part of QR	x	x	x
$F[:,p]$	pivot Vector			x
$F[:,P]$	(pivot) permutation Matrix			x

The following functions are available for the `QR` objects: `size`, `\`. 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 thin or full `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 `full()` which has a named argument `thin`.

Note: `qrfact` 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 than as two separate dense matrices.

The data contained in `QR` or `QRPivoted` can be used to construct the `QRPackedQ` type, which is a compact representation of the rotation matrix:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T)$$

where τ_i is the scale factor and v_i is the projection vector associated with the i^{th} Householder elementary reflector.

The data contained in `QRCompactWY` can be used to construct the `QRCompactWYQ` type, which is a compact representation of the rotation matrix

$$Q = I + YTY^T$$

where `Y` is $m \times r$ lower trapezoidal and `T` is $r \times r$ upper triangular. The *compact WY* representation [Schreiber1989] is not to be confused with the older, *WY* representation [Bischof1987]. (The LAPACK documentation uses `V` in lieu of `Y`.)

qrfact! (`A`, [`pivot=false`])

`qrfact!` is the same as `qrfact()`, but saves space by overwriting the input `A`, instead of creating a copy.

bkfact (`A`) → `BunchKaufman`

Compute the Bunch-Kaufman [Bunch1977] factorization of a real symmetric or complex Hermitian matrix `A` and return a `BunchKaufman` object. The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issym`, `ishermitian`.

bkfact! (`A`) → `BunchKaufman`

`bkfact!` is the same as `bkfact()`, but saves space by overwriting the input `A`, instead of creating a copy.

sqrtn (`A`)

Compute the matrix square root of `A`. If `B = sqrtn(A)`, then `B*B == A` within roundoff error.

`sqrtn` uses a polyalgorithm, computing the matrix square root using Schur factorizations (`schurfact()`) unless it detects the matrix to be Hermitian or real symmetric, in which case it computes the matrix square root from an eigendecomposition (`eigfact()`). In the latter situation for positive definite matrices, the matrix square root has `Real` elements, otherwise it has `Complex` elements.

eig (`A`, [`irange`], [`vl`], [`vu`], [`permute=true`], [`scale=true`]) → `D`, `V`

Computes eigenvalues and eigenvectors of `A`. See `eigfact()` for details on the `balance` keyword argument.

```
julia> eig([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
([1.0, 3.0, 18.0],
 3x3 Array{Float64,2}:
 1.0  0.0  0.0
```

```
0.0  1.0  0.0
0.0  0.0  1.0)
```

`eig` is a wrapper around `eigfact()`, extracting all parts of the factorization to a tuple; where possible, using `eigfact()` is recommended.

`eig(A, B) → D, V`

Computes generalized eigenvalues and vectors of `A` with respect to `B`.

`eig` is a wrapper around `eigfact()`, extracting all parts of the factorization to a tuple; where possible, using `eigfact()` is recommended.

`eigvals(A, [irange], [vl], [vu])`

Returns the eigenvalues of `A`. If `A` is `Symmetric`, `Hermitian` or `SymTridiagonal`, it is possible to calculate only a subset of the eigenvalues by specifying either a `UnitRange` `irange` covering indices of the sorted eigenvalues, or a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

For general non-symmetric 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.

`eigmax(A)`

Returns the largest eigenvalue of `A`.

`eigmin(A)`

Returns the smallest eigenvalue of `A`.

`eigvecs(A, [eigvals], [permute=true], [scale=true]) → Matrix`

Returns a matrix `M` whose columns are the eigenvectors of `A`. (The `k`'th eigenvector can be obtained from the slice `M[:, k]`.) The `permute` and `scale` keywords are the same as for `eigfact()`.

For `SymTridiagonal` matrices, if the optional vector of eigenvalues `eigvals` is specified, returns the specific corresponding eigenvectors.

`eigfact(A, [irange], [vl], [vu], [permute=true], [scale=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 `k`'th eigenvector can be obtained from the slice `F[:vectors][:, k]`.)

The following functions are available for `Eigen` objects: `inv`, `det`.

If `A` is `Symmetric`, `Hermitian` or `SymTridiagonal`, it is possible to calculate only a subset of the eigenvalues by specifying either a `UnitRange` `irange` covering indices of the sorted eigenvalues or a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

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.

`eigfact(A, B) → GeneralizedEigen`

Computes the generalized eigenvalue decomposition of `A` and `B`, returning a `GeneralizedEigen` factorization object `F` which contains the generalized eigenvalues in `F[:values]` and the generalized eigenvectors in the columns of the matrix `F[:vectors]`. (The `k`'th generalized eigenvector can be obtained from the slice `F[:vectors][:, k]`.)

`eigfact!(A[, B])`

Same as `eigfact()`, but saves space by overwriting the input `A` (and `B`), instead of creating a copy.

hessfact (A)

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 `full()`.

hessfact! (A)

`hessfact!` is the same as `hessfact()`, but saves space by overwriting the input A, instead of creating a copy.

schurfact (A) → 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 unitary/orthogonal 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]`.

schurfact! (A)

Computer the Schur factorization of A, overwriting A in the process. See `schurfact()`

schur (A) → Schur[:T], Schur[:Z], Schur[:values]

See `schurfact()`

schurfact (A, B) → 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[:alpha] ./ F[:beta]`.

schur (A, B) → GeneralizedSchur[:S], GeneralizedSchur[:T], GeneralizedSchur[:Q], GeneralizedSchur[:Z]

See `schurfact()`

svdfact (A[, thin=true]) → 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 $A = U * \text{diagm}(S) * Vt$. If `thin` is true, an economy mode decomposition is returned. The algorithm produces Vt and hence Vt is more efficient to extract than V. The default is to produce a thin decomposition.

svdfact! (A[, thin=true]) → SVD

`svdfact!` is the same as `svdfact()`, but saves space by overwriting the input A, instead of creating a copy. If `thin` is true, an economy mode decomposition is returned. The default is to produce a thin decomposition.

svd (A[, thin=true]) → U, S, V

Wrapper around `svdfact` extracting all parts the factorization to a tuple. Direct use of `svdfact` is therefore generally more efficient. Computes the SVD of A, returning U, vector S, and V such that $A == U * \text{diagm}(S) * V'$. If `thin` is true, an economy mode decomposition is returned. The default is to produce a thin decomposition.

svdvals (A)

Returns the singular values of A.

svdvals! (A)

Returns the singular values of A, while saving space by overwriting the input.

svdfact (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 $B = F[:V] * F[:D2] * F[:R0] * F[:Q]'$.

svd (A, B) → U, V, Q, D1, D2, R0

Wrapper around `svdfact` extracting all parts the factorization to a tuple. Direct use of `svdfact` is therefore

generally more efficient. The function returns the generalized SVD of A and B , returning U , V , Q , $D1$, $D2$, and $R0$ such that $A = U \cdot D1 \cdot R0 \cdot Q'$ and $B = V \cdot D2 \cdot R0 \cdot Q'$.

svdvals (A, B)

Return only the singular values from the generalized singular value decomposition of A and B .

triu (M)

Upper triangle of a matrix.

triu! (M)

Upper triangle of a matrix, overwriting M in the process.

tril (M)

Lower triangle of a matrix.

tril! (M)

Lower triangle of a matrix, overwriting M in the process.

diagind ($M[k]$)

A Range giving the indices of the k -th diagonal of the matrix M .

diag ($M[k]$)

The k -th diagonal of a matrix, as a vector. Use `diagm` to construct a diagonal matrix.

diagm ($v[k]$)

Construct a diagonal matrix and place v on the k -th diagonal.

scale (A, b)

scale (b, A)

Scale an array A by a scalar b , returning a new array.

If A is a matrix and b is a vector, then `scale(A,b)` scales each column i of A by $b[i]$ (similar to $A \cdot \text{diagm}(b)$), while `scale(b,A)` scales each row i of A by $b[i]$ (similar to $\text{diagm}(b) \cdot A$), returning a new array.

Note: for large A , `scale` can be much faster than $A \cdot b$ or $b \cdot A$, due to the use of BLAS.

scale! (A, b)

scale! (b, A)

Scale an array A by a scalar b , similar to `scale()` but overwriting A in-place.

If A is a matrix and b is a vector, then `scale!(A,b)` scales each column i of A by $b[i]$ (similar to $A \cdot \text{diagm}(b)$), while `scale!(b,A)` scales each row i of A by $b[i]$ (similar to $\text{diagm}(b) \cdot A$), again operating in-place on A .

Tridiagonal (dl, d, du)

Construct a tridiagonal matrix from the lower diagonal, diagonal, and upper diagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `full()`.

Bidiagonal ($dv, ev, isupper$)

Constructs an upper (`isupper=true`) or lower (`isupper=false`) 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 `full()`.

SymTridiagonal (d, du)

Construct a real symmetric tridiagonal matrix from the diagonal and upper diagonal, respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `full()`.

Woodbury (*A*, *U*, *C*, *V*)

Construct a matrix in a form suitable for applying the Woodbury matrix identity.

rank (*M*)

Compute the rank of a matrix.

norm (*A* [, *p*])

Compute the *p*-norm of a vector or the operator norm of a matrix *A*, defaulting to the *p*=2-norm.

For vectors, *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.

For matrices, valid values of *p* are 1, 2, or `Inf`. (Note that for sparse matrices, *p*=2 is currently not implemented.) Use `vecnorm()` to compute the Frobenius norm.

vecnorm (*A* [, *p*])

For any iterable container *A* (including arrays of any dimension) of numbers, compute the *p*-norm (defaulting to *p*=2) as if *A* were a vector of the corresponding length.

For example, if *A* is a matrix and *p*=2, then this is equivalent to the Frobenius norm.

cond (*M* [, *p*])

Condition number of the matrix *M*, computed using the operator *p*-norm. Valid values for *p* are 1, 2 (default), or `Inf`.

condskeel (*M* [, *x*, *p*])

$$\kappa_S(M, p) = \| |M| |M^{-1}| \|_p$$

$$\kappa_S(M, x, p) = \| |M| |M^{-1}| |x| \|_p$$

Skeel condition number κ_S of the matrix *M*, optionally with respect to the vector *x*, as computed using the operator *p*-norm. *p* is `Inf` by default, if not provided. Valid values for *p* are 1, 2, or `Inf`.

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

trace (*M*)

Matrix trace

det (*M*)

Matrix determinant

logdet (*M*)

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

inv (*M*)

Matrix inverse

pinv (*M*)

Moore-Penrose pseudoinverse

null (*M*)

Basis for nullspace of *M*.

repmat (*A*, *n*, *m*)

Construct a matrix by repeating the given matrix *n* times in dimension 1 and *m* times in dimension 2.

repeat (*A*, *inner* = `Int[]`, *outer* = `Int[]`)

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.

kron (*A*, *B*)

Kronecker tensor product of two vectors or two matrices.

blkdiag (*A*...)

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

linreg (*x*, *y*) → [*a*; *b*]

Linear Regression. Returns *a* and *b* such that $a+b*x$ is the closest line to the given points (*x*, *y*). In other words, this function determines parameters [*a*, *b*] that minimize the squared error between *y* and $a+b*x$.

Example:

```
using PyPlot;
x = float([1:12])
y = [5.5; 6.3; 7.6; 8.8; 10.9; 11.79; 13.48; 15.02; 17.77; 20.81; 22.0; 22.99]
a, b = linreg(x,y) # Linear regression
plot(x, y, "o") # Plot (x,y) points
plot(x, [a+b*i for i in x]) # Plot the line determined by the linear regression
```

linreg (*x*, *y*, *w*)

Weighted least-squares linear regression.

expm (*A*)

Matrix exponential.

lyap (*A*, *C*)

Computes the solution *X* to the continuous Lyapunov equation $AX + XA' + C = 0$, where no eigenvalue of *A* has a zero real part and no two eigenvalues are negative complex conjugates of each other.

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.

issym (*A*) → Bool

Test whether a matrix is symmetric.

isposdef (*A*) → Bool

Test whether a matrix is positive definite.

isposdef! (*A*) → Bool

Test whether a matrix is positive definite, overwriting *A* in the processes.

istril (*A*) → Bool

Test whether a matrix is lower triangular.

istriu (*A*) → Bool

Test whether a matrix is upper triangular.

ishermitian (*A*) → Bool

Test whether a matrix is Hermitian.

transpose (*A*)

The transposition operator (*.*').

ctranspose (*A*)

The conjugate transposition operator (*'*).

eigs (*A*[:, *B*], ; *nev*=6, *which*="LM", *tol*=0.0, *maxiter*=1000, *sigma*=nothing, *ritzvec*=true, *v0*=zeros((0,)))
→ (*d*[:, *v*], *nconv*, *niter*, *nmult*, *resid*)

eigs computes eigenvalues *d* of *A* using Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices.

- *nev*: Number of eigenvalues

- `ncv`: Number of Krylov vectors used in the computation; should satisfy `nev+1 <= ncv <= n` for real symmetric problems and `nev+2 <= ncv <= n` for other problems; default is `ncv = max(20, 2*nev+1)`.
- `which`: type of eigenvalues to compute. See the note below.

<code>which</code>	type of eigenvalues
<code>:LM</code>	eigenvalues of largest magnitude (default)
<code>:SM</code>	eigenvalues of smallest magnitude
<code>:LR</code>	eigenvalues of largest real part
<code>:SR</code>	eigenvalues of smallest real part
<code>:LI</code>	eigenvalues of largest imaginary part (nonsymmetric or complex <code>A</code> only)
<code>:SI</code>	eigenvalues of smallest imaginary part (nonsymmetric or complex <code>A</code> only)
<code>:BE</code>	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric <code>A</code> only)

- `tol`: tolerance (`tol ≤ 0.0` defaults to `DLAMCH('EPS')`)
- `maxiter`: Maximum number of iterations (default = 300)
- `sigma`: Specifies the level shift used in inverse iteration. If `nothing` (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to `sigma` using shift and invert iterations.
- `ritzvec`: Returns the Ritz vectors `v` (eigenvectors) if `true`
- `v0`: starting vector from which to start the iterations

`eigs` returns the `nev` requested eigenvalues in `d`, the corresponding Ritz vectors `v` (only if `ritzvec=true`), the number of converged eigenvalues `nconv`, the number of iterations `niter` and the number of matrix vector multiplications `nmult`, as well as the final residual vector `resid`.

Note: The `sigma` and `which` keywords interact: the description of eigenvalues searched for by `which` do `_not_` necessarily refer to the eigenvalues of `A`, but rather the linear operator constructed by the specification of the iteration mode implied by `sigma`.

<code>sigma</code>	iteration mode	<code>which</code> refers to eigenvalues of
<code>nothing</code>	ordinary (forward)	<code>A</code>
<code>real or complex</code>	inverse with level shift <code>sigma</code>	$(A - \sigma I)^{-1}$

peakflops (`n`; `parallel=false`)

`peakflops` computes the peak flop rate of the computer by using double precision `Base.LinAlg.BLAS.gemm!()`. By default, if no arguments are specified, it multiplies a matrix of size `n × n`, where `n = 2000`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `blas_set_num_threads(n)`.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

2.8.2 BLAS Functions

This module provides wrappers for some of the BLAS functions for linear algebra. Those BLAS functions that overwrite one of the input arrays have names ending in `!`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `Complex128` and `Complex64` arrays.

dot (*n*, *X*, *incx*, *Y*, *incy*)
 Dot product of two vectors consisting of *n* elements of array *X* with stride *incx* and *n* elements of array *Y* with stride *incy*.

dotu (*n*, *X*, *incx*, *Y*, *incy*)
 Dot function for two complex vectors.

dotc (*n*, *X*, *incx*, *U*, *incy*)
 Dot function for two complex vectors conjugating the first vector.

blascopy! (*n*, *X*, *incx*, *Y*, *incy*)
 Copy *n* elements of array *X* with stride *incx* to array *Y* with stride *incy*. Returns *Y*.

nrm2 (*n*, *X*, *incx*)
 2-norm of a vector consisting of *n* elements of array *X* with stride *incx*.

asum (*n*, *X*, *incx*)
 sum of the absolute values of the first *n* elements of array *X* with stride *incx*.

axpy! (*n*, *a*, *X*, *incx*, *Y*, *incy*)
 Overwrite *Y* with $a \cdot X + Y$. Returns *Y*.

scal! (*n*, *a*, *X*, *incx*)
 Overwrite *X* with $a \cdot X$. Returns *X*.

scal (*n*, *a*, *X*, *incx*)
 Returns $a \cdot X$.

syrk! (*uplo*, *trans*, *alpha*, *A*, *beta*, *C*)
 Rank-k update of the symmetric matrix *C* as $\alpha A A^T + \beta C$ or $\alpha A^T A + \beta C$ according to whether *trans* is 'N' or 'T'. When *uplo* is 'U' the upper triangle of *C* is updated ('L' for lower triangle). Returns *C*.

syrk (*uplo*, *trans*, *alpha*, *A*)
 Returns either the upper triangle or the lower triangle, according to *uplo* ('U' or 'L'), of $\alpha A A^T$ or $\alpha A^T A$, according to *trans* ('N' or 'T').

herk! (*uplo*, *trans*, *alpha*, *A*, *beta*, *C*)
 Methods for complex arrays only. Rank-k update of the Hermitian matrix *C* as $\alpha A A^H + \beta C$ or $\alpha A^H A + \beta C$ according to whether *trans* is 'N' or 'T'. When *uplo* is 'U' the upper triangle of *C* is updated ('L' for lower triangle). Returns *C*.

herk (*uplo*, *trans*, *alpha*, *A*)
 Methods for complex arrays only. Returns either the upper triangle or the lower triangle, according to *uplo* ('U' or 'L'), of $\alpha A A^H$ or $\alpha A^H A$, according to *trans* ('N' or 'T').

gbmv! (*trans*, *m*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)
 Update vector *y* as $\alpha A x + \beta y$ or $\alpha A^T x + \beta y$ according to *trans* ('N' or 'T'). The matrix *A* is a general band matrix of dimension *m* by $\text{size}(A, 2)$ with *kl* sub-diagonals and *ku* super-diagonals. Returns the updated *y*.

gbmv (*trans*, *m*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)
 Returns $\alpha A x$ or $\alpha A^T x$ according to *trans* ('N' or 'T'). The matrix *A* is a general band matrix of dimension *m* by $\text{size}(A, 2)$ with *kl* sub-diagonals and *ku* super-diagonals.

sbmv! (*uplo*, *k*, *alpha*, *A*, *x*, *beta*, *y*)
 Update vector *y* as $\alpha A x + \beta y$ where *A* is a symmetric band matrix of order $\text{size}(A, 2)$ 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/>.
 Returns the updated *y*.

sbmv (*uplo, k, alpha, A, x*)

Returns $\alpha A * x$ where A is a symmetric band matrix of order `size(A, 2)` with k super-diagonals stored in the argument A .

sbmv (*uplo, k, A, x*)

Returns $A * x$ where A is a symmetric band matrix of order `size(A, 2)` with k super-diagonals stored in the argument A .

gemm! (*tA, tB, alpha, A, B, beta, C*)

Update C as $\alpha A * B + \beta C$ or the other three variants according to tA (transpose A) and tB . Returns the updated C .

gemm (*tA, tB, alpha, A, B*)

Returns $\alpha A * B$ or the other three variants according to tA (transpose A) and tB .

gemm (*tA, tB, A, B*)

Returns $A * B$ or the other three variants according to tA (transpose A) and tB .

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 (transpose A). Returns the updated y .

gemv (*tA, alpha, A, x*)

Returns $\alpha A * x$ or $\alpha A' * x$ according to tA (transpose A).

gemv (*tA, A, x*)

Returns $A * x$ or $A' * x$ according to tA (transpose A).

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. Returns the updated C .

symm (*side, ul, alpha, A, B*)

Returns $\alpha A * B$ or $\alpha B * A$ according to *side*. A is assumed to be symmetric. Only the *ul* triangle of A is used.

symm (*side, ul, A, B*)

Returns $A * B$ or $B * A$ according to *side*. A is assumed to be symmetric. Only the *ul* triangle of A is used.

symm (*tA, tB, alpha, A, B*)

Returns $\alpha A * B$ or the other three variants according to tA (transpose A) and tB .

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. Returns the updated y .

symv (*ul, alpha, A, x*)

Returns $\alpha A * x$. A is assumed to be symmetric. Only the *ul* triangle of A is used.

symv (*ul, A, x*)

Returns $A * x$. A is assumed to be symmetric. Only the *ul* triangle of A is used.

trmm! (*side, ul, tA, dA, alpha, A, B*)

Update B as $\alpha A * B$ or one of the other three variants determined by *side* (A on left or right) and tA (transpose A). Only the *ul* triangle of A is used. *dA* indicates if A is unit-triangular (the diagonal is assumed to be all ones). Returns the updated B .

trmm (*side, ul, tA, dA, alpha, A, B*)

Returns $\alpha A * B$ or one of the other three variants determined by *side* (A on left or right) and tA (transpose A). Only the *ul* triangle of A is used. *dA* indicates if A is unit-triangular (the diagonal is assumed to be all ones).

trsm! (*side, ul, tA, dA, alpha, A, B*)

Overwrite *B* with the solution to $A * X = \text{alpha} * B$ or one of the other three variants determined by *side* (*A* on left or right of *X*) and *tA* (transpose *A*). Only the *ul* triangle of *A* is used. *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones). Returns the updated *B*.

trsm (*side, ul, tA, dA, alpha, A, B*)

Returns the solution to $A * X = \text{alpha} * B$ or one of the other three variants determined by *side* (*A* on left or right of *X*) and *tA* (transpose *A*). Only the *ul* triangle of *A* is used. *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones).

trmv! (*side, ul, tA, dA, alpha, A, b*)

Update *b* as $\text{alpha} * A * b$ or one of the other three variants determined by *side* (*A* on left or right) and *tA* (transpose *A*). Only the *ul* triangle of *A* is used. *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones). Returns the updated *b*.

trmv (*side, ul, tA, dA, alpha, A, b*)

Returns $\text{alpha} * A * b$ or one of the other three variants determined by *side* (*A* on left or right) and *tA* (transpose *A*). Only the *ul* triangle of *A* is used. *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones).

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* (transpose *A*) and *ul* (triangle of *A* used). *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones). Returns the updated *b*.

trsv (*ul, tA, dA, A, b*)

Returns the solution to $A * x = b$ or one of the other two variants determined by *tA* (transpose *A*) and *ul* (triangle of *A* is used.) *dA* indicates if *A* is unit-triangular (the diagonal is assumed to be all ones).

blas_set_num_threads (*n*)

Set the number of threads the BLAS library should use.

2.9 Constants

OS_NAME

A symbol representing the name of the operating system. Possible values are `:Linux`, `:Darwin` (OS X), or `:Windows`.

ARGS

An array of the command line arguments passed to Julia, as strings.

C_NULL

The C null pointer constant, sometimes used when calling external code.

CPU_CORES

The number of CPU cores in the system.

WORD_SIZE

Standard word size on the current machine, in bits.

VERSION

An object describing which version of Julia is in use.

LOAD_PATH

An array of paths (as strings) where the `require` function looks for code.

2.10 Filesystem

pwd(*⋅*) → String

Get the current working directory.

cd(*dir::String*)

Set the current working directory.

cd(*f*[, *dir*])

Temporarily changes the current working directory (HOME if not specified) and applies function *f* before returning.

mkdir(*path*[, *mode*])

Make a new directory with name *path* and permissions *mode*. *mode* defaults to 0o777, modified by the current file creation mask.

mkpath(*path*[, *mode*])

Create all directories in the given *path*, with permissions *mode*. *mode* defaults to 0o777, modified by the current file creation mask.

symlink(*target*, *link*)

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.

chmod(*path*, *mode*)

Change the permissions mode of *path* to *mode*. Only integer modes (e.g. 0o777) are currently supported.

stat(*file*)

Returns a structure whose fields contain information about the file. The fields of the structure are:

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

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.

ctime(*file*)

Equivalent to *stat*(*file*).*ctime*

mtime(*file*)

Equivalent to *stat*(*file*).*mtime*

filemode(*file*)

Equivalent to *stat*(*file*).*mode*

filesize (*path...*)

Equivalent to `stat(file).size`

uperm (*file*)

Gets the permissions of the owner of the file as a bitfield of

01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see `stat`.

gperm (*file*)

Like `uperm` but gets the permissions of the group owning the file

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

cp (*src::String, dst::String*)

Copy a file from *src* to *dst*.

download (*url*[, *localfile*])

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 need, please use a package that provides the desired functionality instead.

mv (*src::String, dst::String*)

Move a file from *src* to *dst*.

rm (*path::String; recursive=false*)

Delete the file, link, or empty directory at the given path. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

touch (*path::String*)

Update the last-modified timestamp on a file to the current time.

tempname ()

Generate a unique temporary file path.

tempdir ()

Obtain the path of a temporary directory (possibly shared with other processes).

mktemp ()

Returns (*path*, *io*), where *path* is the path of a new temporary file and *io* is an open file object for this path.

mktempdir ()

Create a temporary directory and return its path.

isblockdev (*path*) → Bool

Returns `true` if *path* is a block device, `false` otherwise.

ischardev (*path*) → Bool

Returns `true` if *path* is a character device, `false` otherwise.

isdir (*path*) → Bool

Returns `true` if *path* is a directory, `false` otherwise.

isexecutable (*path*) → Bool

Returns `true` if the current user has permission to execute *path*, `false` otherwise.

isfifo (*path*) → Bool
Returns true if *path* is a FIFO, false otherwise.

isfile (*path*) → Bool
Returns true if *path* is a regular file, false otherwise.

islink (*path*) → Bool
Returns true if *path* is a symbolic link, false otherwise.

ispath (*path*) → Bool
Returns true if *path* is a valid filesystem path, false otherwise.

isreadable (*path*) → Bool
Returns true if the current user has permission to read *path*, false otherwise.

issetgid (*path*) → Bool
Returns true if *path* has the setgid flag set, false otherwise.

issetuid (*path*) → Bool
Returns true if *path* has the setuid flag set, false otherwise.

issocket (*path*) → Bool
Returns true if *path* is a socket, false otherwise.

issticky (*path*) → Bool
Returns true if *path* has the sticky bit set, false otherwise.

iswritable (*path*) → Bool
Returns true if the current user has permission to write to *path*, false otherwise.

homedir () → String
Return the current user's home directory.

dirname (*path::String*) → String
Get the directory part of a path.

basename (*path::String*) → String
Get the file name part of a path.

@__FILE__ () → String
@__FILE__ expands to a string with the absolute path and file name of the script being run. Returns nothing if run from a REPL or an empty string if evaluated by `julia -e <expr>`.

isabspath (*path::String*) → Bool
Determines whether a path is absolute (begins at the root directory).

isdirpath (*path::String*) → Bool
Determines whether a path refers to a directory (for example, ends with a path separator).

joinpath (*parts...*) → String
Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

abspath (*path::String*) → String
Convert a path to an absolute path by adding the current directory if necessary.

normpath (*path::String*) → String
Normalize a path, removing "." and ".." entries.

realpath (*path::String*) → String
Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

expanduser (*path::String*) → String
On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

splitdir (*path::String*) -> (*String*, *String*)

Split a path into a tuple of the directory name and file name.

splitdrive (*path::String*) -> (*String*, *String*)

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.

splitext (*path::String*) -> (*String*, *String*)

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.

2.11 I/O and Network

2.11.1 General I/O

STDOUT

Global variable referring to the standard out stream.

STDERR

Global variable referring to the standard error stream.

STDIN

Global variable referring to the standard input stream.

open (*file_name* [, *read*, *write*, *create*, *truncate*, *append*]) → *IOStream*

Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

open (*file_name* [, *mode*]) → *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:

r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

open (*f::function*, *args...*)

Apply the function *f* to the result of `open(args...)` and close the resulting file descriptor upon completion.

Example: `open(readall, "file.txt")`

IOBuffer () → *IOBuffer*

Create an in-memory I/O stream.

IOBuffer (*size::Int*)

Create a fixed size *IOBuffer*. The buffer will not grow dynamically.

IOBuffer (*string*)

Create a read-only *IOBuffer* on the data underlying the given string

IOBuffer ([*data*] [, *readable*, *writable* [, *maxsize*]])

Create an *IOBuffer*, which may optionally operate on a pre-existing array. If the *readable*/*writable* arguments are given, they restrict whether or not the buffer may be read from or written to respectively. By default the

buffer is readable but not writable. The last argument optionally specifies a size beyond which the buffer may not be grown.

takebuf_array (*b::IOBuffer*)

Obtain the contents of an `IOBuffer` as an array, without copying.

takebuf_string (*b::IOBuffer*)

Obtain the contents of an `IOBuffer` as a string, without copying.

fdio (*[name::String], fd::Integer[, own::Bool]*) → `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.

flush (*stream*)

Commit all currently buffered writes to the given stream.

flush_cstdio ()

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

close (*stream*)

Close an I/O stream. Performs a `flush` first.

write (*stream, x*)

Write the canonical binary representation of a value to the given stream.

read (*stream, type*)

Read a value of the given type from a stream, in canonical binary representation.

read (*stream, type, dims*)

Read a series of values of the given type from a stream, in canonical binary representation. `dims` is either a tuple or a series of integer arguments specifying the size of `Array` to return.

read! (*stream, array::Array*)

Read binary data from a stream, filling in the argument `array`.

readbytes! (*stream, b::Vector{UInt8}, nb=length(b)*)

Read at most `nb` bytes from the stream into `b`, returning the number of bytes read (increasing the size of `b` as needed).

readbytes (*stream, nb=typemax{Int}*)

Read at most `nb` bytes from the stream, returning a `Vector{UInt8}` of the bytes read.

position (*s*)

Get the current position of a stream.

seek (*s, pos*)

Seek a stream to the given position.

seekstart (*s*)

Seek a stream to its beginning.

seekend (*s*)

Seek a stream to its end.

skip (*s, offset*)

Seek a stream relative to the current position.

mark (*s*)

Add a mark at the current position of stream `s`. Returns the marked position.

See also `unmark()`, `reset()`, `ismarked()`

unmark (*s*)

Remove a mark from stream *s*. Returns `true` if the stream was marked, `false` otherwise.

See also `mark()`, `reset()`, `ismarked()`

reset (*s*)

Reset a stream *s* to a previously marked position, and remove the mark. Returns the previously marked position. Throws an error if the stream is not marked.

See also `mark()`, `unmark()`, `ismarked()`

ismarked (*s*)

Returns `true` if stream *s* is marked.

See also `mark()`, `unmark()`, `reset()`

eof (*stream*) → `Bool`

Tests 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.

isreadonly (*stream*) → `Bool`

Determine whether a stream is read-only.

isopen (*stream*) → `Bool`

Determine whether a stream is open (i.e. has not been closed yet). If the connection has been closed remotely (in case of e.g. a socket), `isopen` will return `false` even though buffered data may still be available. Use `eof` to check if necessary.

ntoh (*x*)

Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

hton (*x*)

Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

ltoh (*x*)

Converts the endianness of a value from Little-endian to that used by the Host.

htol (*x*)

Converts the endianness of a value from that used by the Host to Little-endian.

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.

serialize (*stream*, *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.

deserialize (*stream*)

Read a value written by `serialize`.

print_escaped (*io*, *str::String*, *esc::String*)

General escaping of traditional C and Unicode escape sequences, plus any characters in *esc* are also escaped (with a backslash).

print_unescaped (*io*, *s::String*)

General unescaping of traditional C and Unicode escape sequences. Reverse of `print_escaped()`.

print_joined (*io*, *items*, *delim*_[, last])

Print elements of *items* to *io* with *delim* between them. If *last* is specified, it is used as the final delimiter instead of *delim*.

print_shortest (*io*, *x*)

Print the shortest possible representation of number *x* as a floating point number, ensuring that it would parse to the exact same number.

fd (*stream*)

Returns 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.

redirect_stdout ()

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.

redirect_stdout (*stream*)

Replace STDOUT by *stream* for all C and julia level output to STDOUT. Note that *stream* must be a TTY, a Pipe or a TcpSocket.

redirect_stderr (_[stream])

Like `redirect_stdout`, but for STDERR

redirect_stdin (_[stream])

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.

readchomp (*x*)

Read the entirety of *x* as a string but remove trailing newlines. Equivalent to `chomp(readall(x))`.

readdir (_[dir]) → Vector{ByteString}

Returns the files and directories in the directory *dir* (or the current working directory if not given).

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

skipchars (*stream*, *predicate*; *linecomment*::Char)

Advance the stream until before the first character for which *predicate* returns false. For example `skipchars(stream, isspace)` will skip all whitespace. If keyword argument *linecomment* is specified, characters from that character through the end of a line will also be skipped.

countlines (*io*_[, eol::Char])

Read *io* until the end of the stream/file and count the number of non-empty lines. To specify a file pass the file-name as the first argument. EOL markers other than 'n' are supported by passing them as the second argument.

PipeBuffer ()

An IOBuffer that allows reading and performs writes by appending. Seeking and truncating are not supported. See IOBuffer for the available constructors.

PipeBuffer (*data*::Vector{UInt8}_[, maxsize])

Create a PipeBuffer to operate on a data vector, optionally specifying a size beyond which the underlying Array may not be grown.

readavailable (*stream*)

Read all available data on the stream, blocking the task only if no data is available.

2.11.2 Network I/O

connect (*[host]*, *port*) → *TcpSocket*

Connect to the host *host* on port *port*

connect (*path*) → *Pipe*

Connect to the Named Pipe/Domain Socket at *path*

listen (*[addr]*, *port*) → *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.

listen (*path*) → *PipeServer*

Listens on/Creates a Named Pipe/Domain Socket

getaddrinfo (*host*)

Gets the IP address of the *host* (may have to do a DNS lookup)

parseip (*addr*)

Parse a string specifying an IPv4 or IPv6 ip address.

IPv4 (*host::Integer*) → *IPv4*

Returns IPv4 object from ip address formatted as Integer

IPv6 (*host::Integer*) → *IPv6*

Returns IPv6 object from ip address formatted as Integer

nb_available (*stream*)

Returns the number of bytes available for reading before a read from this stream or buffer will block.

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.

listenany (*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.

watch_file (*cb=false*, *s*; *poll=false*)

Watch file or directory *s* and run callback *cb* when *s* is modified. The *poll* parameter specifies whether to use file system event monitoring or polling. The callback function *cb* should accept 3 arguments: (*filename*, *events*, *status*) where *filename* is the name of file that was modified, *events* is an object with boolean fields *changed* and *renamed* when using file system event monitoring, or *readable* and *writable* when using polling, and *status* is always 0. Pass *false* for *cb* to not use a callback function.

poll_fd (*fd*, *seconds::Real*; *readable=false*, *writable=false*)

Poll a file descriptor *fd* for changes in the read or write availability and with a timeout given by the second argument. If the timeout is not needed, use *wait(fd)* instead. The keyword arguments determine which of read and/or write status should be monitored and at least one of them needs to be set to true. The returned value is an object with boolean fields *readable*, *writable*, and *timedout*, giving the result of the polling.

poll_file (*s*, *interval_seconds::Real*, *seconds::Real*)

Monitor a file for changes by polling every *interval_seconds* seconds for *seconds* seconds. A return value of true indicates the file changed, a return value of false indicates a timeout.

bind (*socket::Union(UdpSocket, TcpSocket)*, *host::IPv4*, *port::Integer*)

Bind *socket* to the given *host:port*. Note that *0.0.0.0* will listen on all devices.

send (*socket::UdpSocket*, *host::IPv4*, *port::Integer*, *msg*)

Send *msg* over *socket* to ``*host:port*``.

recv (*socket::UdpSocket*)

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

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. *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.

2.11.3 Text I/O

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.

showcompact (*x*)

Show a more compact representation of a value. This is used for printing array elements. If a new type has a different compact representation, it should overload `showcompact(io, x)` where the first argument is a stream.

showall (*x*)

Similar to `show`, except shows all elements of arrays.

summary (*x*)

Return a string giving a brief description of a value. By default returns `string(typeof(x))`. For arrays, returns strings like “2x2 Float64 Array”.

print (*x*)

Write (to the default output stream) a canonical (un-decorated) text representation of a value if there is one, otherwise call `show`. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

println (*x*)

Print (using `print()`) *x* followed by a newline.

print_with_color (*color::Symbol* [, *io*], *strings...*)

Print strings in a color specified as a symbol, for example `:red` or `:blue`.

info (*msg*)

Display an informational message.

warn (*msg*)

Display a warning.

@printf ([*io::IOStream*], “%Fmt”, *args...*)

Print arg(s) using C `printf()` style format specification string. Optionally, an `IOStream` may be passed as the first argument to redirect output.

@sprintf (“%Fmt”, *args...*)

Return `@printf` formatted output as string.

sprint (*f::Function*, *args...*)

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.

showerror (*io*, *e*)

Show a descriptive representation of an exception object.

dump (*x*)
Show all user-visible structure of a value.

xdump (*x*)
Show all structure of a value, including all fields of objects.

readall (*stream::IO*)
Read the entire contents of an I/O stream as a string.

readall (*filename::String*)
Open *filename*, read the entire contents as a string, then close the file. Equivalent to `open(readall, filename)`.

readline (*stream=STDIN*)
Read a single line of text, including a trailing newline character (if one is reached before the end of the input), from the given *stream* (defaults to `STDIN`),

readuntil (*stream, delim*)
Read a string, up to and including the given delimiter byte.

readlines (*stream*)
Read all lines as an array.

eachline (*stream*)
Create an iterable object that will yield each line from a stream.

readdlm (*source, delim::Char, T::Type, eol::Char; header=false, skipstart=0, use_mmap, ignore_invalid_chars=false, quotes=true, dims, comments=true, 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 `ASCIIString`, `String`, 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 *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 *ignore_invalid_chars* is `true`, bytes in *source* with invalid character encoding will be ignored. Otherwise an error is thrown indicating the offending character position.

If *quotes* is `true`, column 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.

readdlm (*source, delim::Char, eol::Char; options...*)
If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a cell array of numbers and strings is returned.

readdlm (*source, delim::Char, T::Type; options...*)
The end of line delimiter is taken as `\n`.

readdlm (*source*, *delim::Char*; *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 cell array of numbers and strings is returned.

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`.

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 cell array of numbers and strings is returned.

writedlm (*f*, *A*, *delim='t'*)

Write *A* (either an array type 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 `String`).

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))`.

readcsv (*source*, [*T::Type*]; *options...*)

Equivalent to `readdlm` with *delim* set to comma.

writcsv (*filename*, *A*)

Equivalent to `writedlm` with *delim* set to comma.

Base64Pipe (*ostream*)

Returns 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 `Base64Pipe` stream is necessary to complete the encoding (but does not close *ostream*).

base64 (*writefunc*, *args...*)

base64 (*args...*)

Given a write-like function *writefunc*, which takes an I/O stream as its first argument, `base64(writefunc, args...)` calls *writefunc* to write *args...* to a base64-encoded string, and returns the string. `base64(args...)` is equivalent to `base64(write, args...)`: it converts its arguments into bytes using the standard `write` functions and returns the base64-encoded string.

2.11.4 Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function `display(x)` to request the richest available multimedia display of a Julia object *x* (with a plain-text fallback).
- Overloading `writemime` 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 `Display` 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).

display (*x*)

display (*d::Display*, *x*)

display (*mime*, *x*)

display (*d::Display*, *mime*, *x*)

Display *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.

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::String* (for MIME types with text-based storage, such as `text/html` or `application/postscript`) or *x::Vector{UInt8}* (for binary MIME types).

redisplay (*x*)

redisplay (*d::Display*, *x*)

redisplay (*mime*, *x*)

redisplay (*d::Display*, *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.

displayable (*mime*) → `Bool`

displayable (*d::Display*, *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.

writemime (*stream*, *mime*, *x*)

The display functions ultimately call `writemime` in order to write an object *x* as a given *mime* type to a given I/O *stream* (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 `writemime` method for *T*, via: `writemime(stream, ::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 *stream*. (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 `writemime(stream, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `Display` (such as `IJulia`). As usual, be sure to import `Base.writemime` in order to add new methods to the built-in Julia function `writemime`.

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.

mimewritable (*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 `writemime` function for `typeof(x)`.)

reprmime (*mime*, *x*)

Returns a `String` or `Vector{UInt8}` containing the representation of *x* in the requested *mime* type, as written by `writemime` (throwing a `MethodError` if no appropriate `writemime` is available). A `String` 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 `istext(mime)` returns whether or not Julia treats a given *mime* type as text.)

As a special case, if *x* is a `String` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `reprmime` function assumes that *x* is already in the requested *mime* format and simply returns *x*.

stringmime (*mime*, *x*)

Returns a `String` containing the representation of *x* in the requested *mime* type. This is similar to `reprmime`

except that binary data is base64-encoded as an ASCII string.

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 `Display`. 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 `reprmime(mime, x)`. A `MethodError` should be thrown if `x` cannot be displayed as that MIME type; this is automatic if one calls `reprmime`. Finally, one should define a function `display(d::D, x)` that queries `mimewritable(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:

pushdisplay (*d::Display*)

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`).

popdisplay ()

popdisplay (*d::Display*)

Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

TextDisplay (*stream*)

Returns a `TextDisplay <: Display`, which can display any object as the text/plain MIME type (only), writing the text representation to the given I/O stream. (The text representation is the same as the way an object is printed in the Julia REPL.)

istext (*m::MIME*)

Determine whether a MIME type is text data.

2.11.5 Memory-mapped I/O

mmap_array (*type, dims, stream[, offset]*)

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 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 specifying the size of the array.

The file is passed via the `stream` argument. 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.

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 example, the following code:

```
# Create a file for mmaping
# (you could alternatively use mmap_array to do this step, too)
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
```

```

write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin") # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = mmap_array{Int, (m,n), s}

```

creates a `m-by-n Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size—32 bit or 64 bit—and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

mmap_bitarray (`[type]`, `dims`, `stream`, `[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_array()`, but the byte representation is different. The `type` parameter is optional, and must be `Bool` if given.

Example: `B = mmap_bitarray((25,30000), s)`

This would create a 25-by-30000 `BitArray`, linked to the file associated with stream `s`.

msync (`array`)

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

msync (`ptr`, `len`, `[flags]`)

Forces synchronization of the `mmap()`ed memory region from `ptr` to `ptr+len`. `Flags` defaults to `MS_SYNC`, but can be a combination of `MS_ASYNC`, `MS_SYNC`, or `MS_INVALIDATE`. See your platform man page for specifics. The `flags` argument is not valid on Windows.

You may not need to call `msync`, because synchronization is performed at intervals automatically by the operating system. However, you can call this directly if, for example, you are concerned about losing the result of a long-running calculation.

MS_ASYNC

Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

MS_SYNC

Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

MS_INVALIDATE

Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

mmap (`len`, `prot`, `flags`, `fd`, `offset`)

Low-level interface to the `mmap` system call. See the man page.

munmap (`pointer`, `len`)

Low-level interface for unmapping memory (see the man page). With `mmap_array()` you do not need to call this directly; the memory is unmapped for you when the array goes out of scope.

2.12 Punctuation

Extended documentation for mathematical symbols & functions is [here](#).

symbol	meaning
@m	invoke macro m; followed by space-separated expressions
!	prefix “not” operator
a! ()	at the end of a function name, ! indicates that a function modifies its argument(s)
#	begin single line comment
#=	begin multi-line comment (these are nestable)
=#	end multi-line comment
\$	xor operator, string and expression interpolation
%	remainder operator
^	exponent operator
&	bitwise and
*	multiply, or matrix multiply
()	the empty tuple
~	bitwise not operator
\	backslash operator
'	complex transpose operator A^H
a[]	array indexing
[,]	vertical concatenation
[;]	also vertical concatenation
[]	with space-separated expressions, horizontal concatenation
T{ }	parametric type instantiation
{ }	construct a cell array (deprecated in 0.4 in favor of Any[])
;	statement separator
,	separate function arguments or tuple components
?	3-argument conditional operator (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 or type
.	access named fields in objects or names inside modules, also prefixes elementwise operators
a:b	range a, a+1, a+2, ..., b
a:s:b	range a, a+s, a+2s, ..., b
:	index an entire dimension (1:end)
::	type annotation, depending on context
:()	quoted expression
:a	symbol a

2.13 Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. For many users, sorting in standard ascending order, letting Julia pick reasonable default algorithms will be sufficient:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

You can easily sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

To sort an array in-place, use the “bang” version of the sort function:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array’s indices that puts the array into sorted order:

```
julia> v = randn(5)
5-element Array{Float64,1}:
 0.297288
 0.382396
-0.597634
-0.0104452
-0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
-0.839027
-0.597634
-0.0104452
 0.297288
 0.382396
```

Arrays can easily be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
-0.0104452
 0.297288
 0.382396
-0.597634
-0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
-0.839027
```

```
-0.597634
 0.382396
 0.297288
-0.0104452
```

Reasonable sorting algorithms are used by default, but you can choose other algorithms as well:

```
julia> sort(v, alg=InsertionSort)
5-element Array{Float64,1}:
-0.839027
-0.597634
-0.0104452
 0.297288
 0.382396
```

2.13.1 Sorting Functions

sort! (*v*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

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.

sort (*v*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Variant of `sort!` that returns a sorted copy of *v* leaving *v* itself unmodified.

sort (*A*, *dim*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Sort a multidimensional array *A* along the given dimension.

sortperm (*v*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Return a permutation vector of indices of *v* that puts it in sorted order. Specify *alg* to choose a particular sorting algorithm (see [Sorting Algorithms](#)). `MergeSort` is used by default, and since it is stable, the resulting permutation will be the lexicographically first one that puts the input array into sorted order – i.e. indices of equal elements appear in ascending order. If you choose a non-stable sorting algorithm such as `QuickSort`, a different permutation that puts the array into order may be returned. The order is specified using the same keywords as `sort!`.

sortrows (*A*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Sort the rows of matrix *A* lexicographically.

sortcols (*A*, [*alg*=<algorithm>], [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Sort the columns of matrix *A* lexicographically.

2.13.2 Order-Related Functions

isorted (*v*, [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Test whether a vector is in sorted order. The *by*, *lt* and *rev* keywords modify what order is considered to be sorted just as they do for `sort`.

searchsorted (*a*, *x*, [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Returns the range of indices of *a* which compare as equal to *x* according to the order specified by the *by*, *lt* and *rev* keywords, assuming that *a* is already sorted in that order. Returns an empty range located at the insertion point if *a* does not contain values equal to *x*.

searchsortedfirst (*a*, *x*, [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Returns the index of the first value in *a* greater than or equal to *x*, according to the specified order. Returns `length(a) + 1` if *x* is greater than all values in *a*.

searchsortedlast (*a*, *x*, [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Returns the index of the last value in *a* less than or equal to *x*, according to the specified order. Returns 0 if *x* is less than all values in *a*.

select! (*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. 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 `select!` does not fully sort the input array, but does leave the returned elements where they would be if the array were fully sorted.

select (*v*, *k*, [*by*=<transform>], [*lt*=<comparison>], [*rev*=false])

Variant of `select!` which copies *v* before partially sorting it, thereby returning the same thing as `select!` but leaving *v* unmodified.

2.13.3 Sorting Algorithms

There are currently three sorting algorithms available in base Julia:

- `InsertionSort`
- `QuickSort`
- `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.

`MergeSort` is an $O(n \log n)$ stable sorting algorithm but is not in-place – it requires a temporary array of equal size to the input array – and is typically not quite as fast as `QuickSort`. It is the default algorithm for non-numeric data.

The sort functions select a reasonable default algorithm, depending on the type of the array to be sorted. To force a specific algorithm to be used for `sort` or other sorting functions, supply `alg=<algorithm>` as a keyword argument after the array to be sorted.

2.14 Package Manager Functions

All package manager functions are defined in the `Pkg` module. None of the `Pkg` module's functions are exported; to use them, you'll need to prefix each function call with an explicit `Pkg.`, e.g. `Pkg.status()` or `Pkg.dir()`.

dir() → String

Returns the absolute path of the package directory. This defaults to `joinpath(homedir(), ".julia")` on all platforms (i.e. `~/ .julia` in UNIX shell syntax). If the `JULIA_PKGDIR` environment variable is set, that path is used instead. If `JULIA_PKGDIR` is a relative path, it is interpreted relative to whatever the current working directory is.

dir(names...) → String

Equivalent to `normpath(Pkg.dir(), names...)` – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, `Pkg.dir(pkg)` returns the path to the package `pkg`.

init (*meta::String=DEFAULT_META, branch::String=META_BRANCH*)

Initialize `Pkg.dir()` as a package directory. This will be done automatically when the `JULIA_PKGDIR` is not set and `Pkg.dir()` uses its default value. As part of this process, clones a local METADATA git repository from the site and branch specified by its arguments, which are typically not provided. Explicit (non-default) arguments can be used to support a custom METADATA setup.

resolve ()

Determines an optimal, consistent set of package versions to install or upgrade to. The optimal set of package versions is based on the contents of `Pkg.dir("REQUIRE")` and the state of installed packages in `Pkg.dir()`, Packages that are no longer required are moved into `Pkg.dir(".trash")`.

edit ()

Opens `Pkg.dir("REQUIRE")` in the editor specified by the `VISUAL` or `EDITOR` environment variables; when the editor command returns, it runs `Pkg.resolve()` to determine and install a new optimal set of installed package versions.

add (*pkg, vers...*)

Add a requirement entry for `pkg` to `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`. If `vers` are given, they must be `VersionNumber` objects and they specify acceptable version intervals for `pkg`.

rm (*pkg*)

Remove all requirement entries for `pkg` from `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`.

clone (*url[, pkg]*)

Clone a package directly from the git URL `url`. The package does not need to be a registered in `Pkg.dir("METADATA")`. The package repo is cloned by the name `pkg` if provided; if not provided, `pkg` is determined automatically from `url`.

clone (*pkg*)

If `pkg` has a URL registered in `Pkg.dir("METADATA")`, clone it from that URL on the default branch. The package does not need to have any registered versions.

available () → Vector{ASCIIString}

Returns the names of available packages.

available (*pkg*) → Vector{VersionNumber}

Returns the version numbers available for package `pkg`.

installed () → Dict{ASCIIString,VersionNumber}

Returns a dictionary mapping installed package names to the installed version number of each package.

installed (*pkg*) → Nothing | VersionNumber

If `pkg` is installed, return the installed version number, otherwise return `nothing`.

status ()

Prints out a summary of what packages are installed and what version and state they're in.

update ()

Update package the metadata repo – kept in `Pkg.dir("METADATA")` – then update any fixed packages that can safely be pulled from their origin; then call `Pkg.resolve()` to determine a new optimal set of packages versions.

checkout (*pkg[, branch="master"]*)

Checkout the `Pkg.dir(pkg)` repo to the branch `branch`. Defaults to checking out the “master” branch. To go back to using the newest compatible released version, use `Pkg.free(pkg)`

pin (*pkg*)

Pin `pkg` at the current version. To go back to using the newest compatible released version, use `Pkg.free(pkg)`

pin (*pkg*, *version*)

Pin *pkg* at registered version *version*.

free (*pkg*)

Free the package *pkg* to be managed by the package manager again. It calls `Pkg.resolve()` to determine optimal package versions after. This is an inverse for both `Pkg.checkout` and `Pkg.pin`.

build ()

Run the build scripts for all installed packages in depth-first recursive order.

build (*pkgs...*)

Run the build script in “`deps/build.jl`” for each package in *pkgs* and all of their dependencies in depth-first recursive order. This is called automatically by `Pkg.resolve()` on all installed or updated packages.

generate (*pkg*, *license*)

Generate a new package named *pkg* with one of these license keys: “MIT” or “BSD”. If you want to make a package with a different license, you can edit it afterwards. Generate creates a git repo at `Pkg.dir(pkg)` for the package and inside it `LICENSE.md`, `README.md`, the julia entrypoint `$pkg/src/$pkg.jl`, and a travis test file, `.travis.yml`.

register (*pkg*[, *url*])

Register *pkg* at the git URL *url*, defaulting to the configured origin URL of the git repo `Pkg.dir(pkg)`.

tag (*pkg*[, *ver*[, *commit*]])

Tag *commit* as version *ver* of package *pkg* and create a version entry in METADATA. If not provided, *commit* defaults to the current commit of the *pkg* repo. If *ver* is one of the symbols `:patch`, `:minor`, `:major` the next patch, minor or major version is used. If *ver* is not provided, it defaults to `:patch`.

publish ()

For each new package version tagged in METADATA not already published, make sure that the tagged package commits have been pushed to the repo at the registered URL for the package and if they all have, open a pull request to METADATA.

test ()

Run the tests for all installed packages ensuring that each package’s test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`.

test (*pkgs...*)

Run the tests for each package in *pkgs* ensuring that each package’s test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`.

2.15 Graphics

The `Base.Graphics` interface is an abstract wrapper; specific packages (e.g., Cairo and Tk/Gtk) implement much of the functionality.

2.15.1 Geometry

Vec2 (*x*, *y*)

Creates a point in two dimensions

BoundingBox (*xmin*, *xmax*, *ymin*, *ymax*)

Creates a box in two dimensions with the given edges

BoundingBox (*objs...*)

Creates a box in two dimensions that encloses all objects

width (*obj*)

Computes the width of an object

height (*obj*)

Computes the height of an object

xmin (*obj*)

Computes the minimum x-coordinate contained in an object

xmax (*obj*)

Computes the maximum x-coordinate contained in an object

ymin (*obj*)

Computes the minimum y-coordinate contained in an object

ymax (*obj*)

Computes the maximum y-coordinate contained in an object

diagonal (*obj*)

Return the length of the diagonal of an object

aspect_ratio (*obj*)

Compute the height/width of an object

center (*obj*)

Return the point in the center of an object

xrange (*obj*)

Returns a tuple (`xmin(obj)`, `xmax(obj)`)

yrange (*obj*)

Returns a tuple (`ymin(obj)`, `ymax(obj)`)

rotate (*obj, angle, origin*) → *newobj*

Rotates an object around origin by the specified angle (radians), returning a new object of the same type. Because of type-constancy, this new object may not always be a strict geometric rotation of the input; for example, if *obj* is a `BoundingBox` the return is the smallest `BoundingBox` that encloses the rotated input.

shift (*obj, dx, dy*)

Returns an object shifted horizontally and vertically by the indicated amounts

***** (*obj, s::Real*)

Scale the width and height of a graphics object, keeping the center fixed

+ (*bb1::BoundingBox, bb2::BoundingBox*) → `BoundingBox`

Returns the smallest box containing both boxes

& (*bb1::BoundingBox, bb2::BoundingBox*) → `BoundingBox`

Returns the intersection, the largest box contained in both boxes

deform (*bb::BoundingBox, dxmin, dxmax, dymin, dymax*)

Returns a bounding box with all edges shifted by the indicated amounts

isinside (*bb::BoundingBox, x, y*)

True if the given point is inside the box

isinside (*bb::BoundingBox, point*)

True if the given point is inside the box

2.16 Unit and Functional Testing

The `Test` module contains macros and functions related to testing. A default handler is provided to run the tests, and a custom one can be provided by the user by using the `registerhandler()` function.

2.16.1 Overview

To use the default handler, the macro `@test()` can be used directly:

```
julia> using Base.Test

julia> @test 1 == 1

julia> @test 1 == 0
ERROR: test failed: 1 == 0
  in error at error.jl:21
  in default_handler at test.jl:19
  in do_test at test.jl:39

julia> @test error("This is what happens when a test fails")
ERROR: test error during error("This is what happens when a test fails")
This is what happens when a test fails
  in error at error.jl:21
  in anonymous at test.jl:62
  in do_test at test.jl:37
```

As seen in the examples above, failures or errors will print the abstract syntax tree of the expression in question.

Another macro is provided to check if the given expression throws an exception of type `extype`, `@test_throws()`:

```
julia> @test_throws ErrorException error("An error")

julia> @test_throws BoundsError error("An error")
ERROR: test failed: error("An error")
  in error at error.jl:21
  in default_handler at test.jl:19
  in do_test_throws at test.jl:55

julia> @test_throws DomainError throw(DomainError())

julia> @test_throws DomainError throw(EOFError())
ERROR: test failed: throw(EOFError())
  in error at error.jl:21
  in default_handler at test.jl:19
  in do_test_throws at test.jl:55
```

As floating point comparisons can be imprecise, two additional macros exist taking in account small numerical errors:

```
julia> @test_approx_eq 1. 0.999999999
ERROR: assertion failed: |1.0 - 0.999999999| < 2.220446049250313e-12
  1.0 = 1.0
  0.999999999 = 0.999999999
  in test_approx_eq at test.jl:75
  in test_approx_eq at test.jl:80

julia> @test_approx_eq 1. 0.9999999999999999
```

```
julia> @test_approx_eq_eps 1. 0.999 1e-2

julia> @test_approx_eq_eps 1. 0.999 1e-3
ERROR: assertion failed: |1.0 - 0.999| <= 0.001
 1.0 = 1.0
 0.999 = 0.999
 difference = 0.00100000000000000009 > 0.001
in error at error.jl:22
in test_approx_eq at test.jl:68
```

2.16.2 Handlers

A handler is a function defined for three kinds of arguments: Success, Failure, Error:

```
# The definition of the default handler
default_handler(r::Success) = nothing
default_handler(r::Failure) = error("test failed: $(r.expr)")
default_handler(r::Error)   = rethrow(r)
```

A different handler can be used for a block (with `with_handler()`):

```
julia> using Base.Test

julia> custom_handler(r::Test.Success) = println("Success on $(r.expr)")
custom_handler (generic function with 1 method)

julia> custom_handler(r::Test.Failure) = error("Error on custom handler: $(r.expr)")
custom_handler (generic function with 2 methods)

julia> custom_handler(r::Test.Error) = rethrow(r)
custom_handler (generic function with 3 methods)

julia> Test.with_handler(custom_handler) do
    @test 1 == 1
    @test 1 != 1
end
Success on :((1==1))
ERROR: Error on custom handler: :((1!=1))
in error at error.jl:21
in custom_handler at none:1
in do_test at test.jl:39
in anonymous at no file:3
in task_local_storage at task.jl:28
in with_handler at test.jl:24
```

2.16.3 Macros

@test (*ex*)

Test the expression *ex* and calls the current handler to handle the result.

@test_throws (*extype, ex*)

Test that the expression *ex* throws an exception of type *extype* and calls the current handler to handle the result.

@test_approx_eq (*a, b*)

Test two floating point numbers *a* and *b* for equality taking in account small numerical errors.

@test_approx_eq_eps (*a*, *b*, *tol*)

Test two floating point numbers *a* and *b* for equality taking in account a margin of tolerance given by *tol*.

2.16.4 Functions

with_handler (*f*, *handler*)

Run the function *f* using the *handler* as the handler.

2.17 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()`.

runtests (*[tests=["all"]*, *[numcores=iceil(CPU_CORES/2)]*)

Run the Julia unit tests listed in *tests*, which can be either a string or an array of strings, using *numcores* processors.

2.18 C Interface

ccall (*((symbol, library) or fptr, RetType, (ArgType1, ...), ArgVar1, ...)*)

Call function in C-exported shared library, specified by *(function name, library)* tuple, where each component is a String or :Symbol. Alternatively, `ccall` may be used to call a function pointer returned by `dlsym`, but note that this usage is generally discouraged to facilitate future static compilation. Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

cglobal (*((symbol, library) or ptr[, Type=Void])*)

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{Void}` if no *Type* argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

cfunction (*(fun::Function, RetType::Type, (ArgTypes...))*)

Generate C-callable function pointer from Julia function. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

For example:

```
function foo()
    # body

    retval::Float64
end

bar = cfunction(foo, Float64, ())
```

dlopen (*(libfile::String[, flags::Integer])*)

Load a shared library, returning an opaque handle.

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 is `RTLD_LAZY | RTLD_DEEPBIND | RTLD_LOCAL`. An important usage of these flags, on POSIX platforms, is

to specify `RTLD_LAZY` | `RTLD_DEEPBIND` | `RTLD_GLOBAL` in order for the library's symbols to be available for usage in other shared libraries, in situations where there are dependencies between shared libraries.

dlopen_e (*libfile::String* [, *flags::Integer*])

Similar to `dlopen()`, except returns a `NULL` pointer instead of raising errors.

RTLD_DEEPBIND

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_FIRST

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_GLOBAL

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_LAZY

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_LOCAL

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NODELETE

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NOLOAD

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NOW

Enum constant for `dlopen()`. See your platform man page for details, if applicable.

dlsym (*handle, sym*)

Look up a symbol from a shared library handle, return callable function pointer on success.

dlsym_e (*handle, sym*)

Look up a symbol from a shared library handle, silently return `NULL` pointer on lookup failure.

dlclose (*handle*)

Close shared library referenced by handle.

find_library (*names, locations*)

Searches for the first library in *names* 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.

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.

c_malloc (*size::Integer*) → `Ptr{Void}`

Call `malloc` from the C standard library.

c_calloc (*num::Integer, size::Integer*) → `Ptr{Void}`

Call `calloc` from the C standard library.

c_realloc (*addr::Ptr, size::Integer*) → `Ptr{Void}`

Call `realloc` from the C standard library.

c_free (*addr::Ptr*)

Call `free` from the C standard library.

unsafe_load (*p::Ptr{T}, i::Integer*)

Load a value of type *T* from the address of the *i*th element (1-indexed) starting at *p*. This is equivalent to the C expression `p[i-1]`.

unsafe_store! (*p::Ptr{T}, x, i::Integer*)

Store a value of type *T* to the address of the *i*th element (1-indexed) starting at *p*. This is equivalent to the C expression `p[i-1] = x`.

unsafe_copy! (*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.

unsafe_copy! (*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).

copy! (*dest, src*)

Copy all elements from collection *src* to array *dest*. Returns *dest*.

copy! (*dest, do, src, so, N*)

Copy *N* elements from collection *src* starting at offset *so*, to array *dest* starting at offset *do*. Returns *dest*.

pointer (*a[, index]*)

Get the native address of an array or string element. Be careful to ensure that a julia reference to *a* exists as long as this pointer will be used.

pointer (*type, int*)

Convert an integer to a pointer of the specified element type.

pointer_to_array (*p, dims[, own]*)

Wrap a native pointer as a Julia Array object. The pointer element type determines the array element type. *own* optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

pointer_from_objref (*obj*)

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.

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.

disable_sigint (*f::Function*)

Disable Ctrl-C handler during execution of a function, for calling external code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

reenable_sigint (*f::Function*)

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

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.

systemerror (*sysfunc*, *iftrue*)

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `bool` is `true`

strerror (*n*)

Convert a system call error code to a descriptive string

Cchar

Equivalent to the native `char` c-type

Cuchar

Equivalent to the native unsigned `char` c-type (`UInt8`)

Cshort

Equivalent to the native signed `short` c-type (`Int16`)

Cushort

Equivalent to the native unsigned `short` c-type (`UInt16`)

Cint

Equivalent to the native signed `int` c-type (`Int32`)

Cuint

Equivalent to the native unsigned `int` c-type (`UInt32`)

Clong

Equivalent to the native signed `long` c-type

Culong

Equivalent to the native unsigned `long` c-type

Clonglong

Equivalent to the native signed `long long` c-type (`Int64`)

Culonglong

Equivalent to the native unsigned `long long` c-type (`UInt64`)

Csize_t

Equivalent to the native `size_t` c-type (`UInt`)

Cssize_t

Equivalent to the native `ssize_t` c-type

Cptrdiff_t

Equivalent to the native `ptrdiff_t` c-type (`Int`)

Coff_t

Equivalent to the native `off_t` c-type

Cwchar_t

Equivalent to the native `wchar_t` c-type (`Int32`)

Cfloat

Equivalent to the native `float` c-type (`Float32`)

Cdouble

Equivalent to the native `double` c-type (`Float64`)

2.19 Profiling

`@profile()`

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

`clear()`

Clear any existing backtraces from the internal buffer.

`print([io::IO = STDOUT], [data::Vector]; format = :tree, C = false, combine = true, cols = tty_cols())`

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. `format` can be `:tree` or `:flat`. If `C==true`, backtraces from C and Fortran code are shown. `combine==true` merges instruction pointers that correspond to the same line of code. `cols` controls the width of the display.

`print([io::IO = STDOUT], data::Vector, lidict::Dict; format = :tree, combine = true, cols = tty_cols())`

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.

`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)`.

`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.

`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.

`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.

- [Clarke61] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.
- [Bischof1987] 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
- [Schreiber1989] 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
- [Bunch1977] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, *Mathematics of Computation* 31:137 (1977), 163-179. [url](#).

Symbols

$\ast()$ (built-in function), 259
 $\ast()$ (in module Base), 234
 $+$ (in module Base), 234
 $-()$ (in module Base), 234
 $.$
 $\quad =()$ (in module Base), 236
 $\cdot\ast()$ (in module Base), 234
 $\cdot+$ (in module Base), 234
 $\cdot-$ (in module Base), 234
 $\cdot/()$ (in module Base), 234
 $\cdot==()$ (in module Base), 236
 $\cdot\geq()$ (in module Base), 236
 $\cdot\leq()$ (in module Base), 236
 $\cdot\neq()$ (in module Base), 236
 $\cdot^{\wedge}()$ (in module Base), 234
 $\cdot\backslash()$ (in module Base), 234
 $\cdot>()$ (in module Base), 219, 236
 $\cdot>=()$ (in module Base), 236
 $\cdot<()$ (in module Base), 236
 $\cdot<=()$ (in module Base), 236
 $/()$ (in module Base), 234
 $//()$ (in module Base), 235
 $:$ (in module Base), 235
 $==()$ (in module Base), 236
 $===()$ (in module Base), 213, 236
 $\$(\text{in module Base}), 237$
 $\%()$ (in module Base), 235
 $\&()$ (in module Base), 237
 γ (in module Base), 255
 ϕ (in module Base), 255
 π (in module Base), 255
 $\cap()$ (in module Base), 231
 $\cdot()$ (in module Base), 277
 $\cup()$ (in module Base), 231
 $\equiv()$ (in module Base), 213, 236
 $\geq()$ (in module Base), 236
 $\in()$ (in module Base), 225
 $\leq()$ (in module Base), 236
 $\ni()$ (in module Base), 225

$\neq()$ (in module Base), 236
 $\neq()$ (in module Base), 236
 $\nexists()$ (in module Base), 225
 $\notin()$ (in module Base), 225
 $\nsubseteq()$ (in module Base), 229
 $\subseteq()$ (in module Base), 229, 232
 $\subsetneq()$ (in module Base), 229
 $\div()$ (in module Base), 234
 $\times()$ (in module Base), 277
 $\wedge()$ (built-in function), 259
 $\wedge()$ (in module Base), 234
 $\sim()$ (in module Base), 237
 $\backslash()$ (in module Base), 234
 $l()$ (in module Base), 237
 $l>()$ (in module Base), 217, 219
 $>()$ (in module Base), 236
 $>=()$ (in module Base), 236
 $>>()$ (in module Base), 219, 235
 $>>>()$ (in module Base), 235
 $<()$ (in module Base), 236
 $<:()$ (in module Base), 215
 $<=()$ (in module Base), 236
 $<<()$ (in module Base), 235

A

$A_ldiv_Bc()$ (in module Base), 237
 $A_ldiv_Bt()$ (in module Base), 237
 A_mul_B
 $\quad ()$ (in module Base), 237
 $A_mul_Bc()$ (in module Base), 237
 $A_mul_Bt()$ (in module Base), 237
 $A_rdiv_Bc()$ (in module Base), 237
 $A_rdiv_Bt()$ (in module Base), 237
 $abs()$ (in module Base), 242
 $abs2()$ (in module Base), 242
 $abspath()$ (in module Base), 292
 $Ac_ldiv_B()$ (in module Base), 237
 $Ac_ldiv_Bc()$ (in module Base), 237
 $Ac_mul_B()$ (in module Base), 238
 $Ac_mul_Bc()$ (in module Base), 238
 $Ac_rdiv_B()$ (in module Base), 238

`Ac_rdiv_Bc()` (in module `Base`), 238
`accept()` (in module `Base`), 297
`acos()` (in module `Base`), 239
`acosd()` (in module `Base`), 239
`acosh()` (in module `Base`), 240
`acot()` (in module `Base`), 239
`acotd()` (in module `Base`), 240
`acoth()` (in module `Base`), 240
`acsc()` (in module `Base`), 239
`acscd()` (in module `Base`), 240
`acsch()` (in module `Base`), 240
`add()` (in module `Base.Pkg`), 308
`addprocs()` (in module `Base`), 274
`airy()` (in module `Base`), 244
`airyai()` (in module `Base`), 244
`airyaiprime()` (in module `Base`), 244
`airybi()` (in module `Base`), 244
`airybiprime()` (in module `Base`), 244
`airyprime()` (in module `Base`), 244
`airyx()` (in module `Base`), 245
`all`
 `()` (in module `Base`), 227
`all()` (in module `Base`), 227
`angle()` (in module `Base`), 243
`any`
 `()` (in module `Base`), 227
`any()` (in module `Base`), 227
`append`
 `()` (in module `Base`), 232
`applicable()` (in module `Base`), 217
`apply()` (in module `Base`), 217
`apropos()` (in module `Base`), 212
`ARGS` (in module `Base`), 289
`ArgumentError` (in module `Base`), 221
`Array()` (in module `Base`), 265
`ascii()` (built-in function), 259
`asec()` (in module `Base`), 239
`asecd()` (in module `Base`), 240
`asech()` (in module `Base`), 240
`asin()` (in module `Base`), 239
`asind()` (in module `Base`), 239
`asinh()` (in module `Base`), 240
`aspect_ratio()` (in module `Base.Graphics`), 310
`assert()` (in module `Base`), 221
`asum()` (in module `Base.LinAlg.BLAS`), 287
`At_ldiv_B()` (in module `Base`), 238
`At_ldiv_Bt()` (in module `Base`), 238
`At_mul_B()` (in module `Base`), 238
`At_mul_Bt()` (in module `Base`), 238
`At_rdiv_B()` (in module `Base`), 238
`At_rdiv_Bt()` (in module `Base`), 238
`atan()` (in module `Base`), 239
`atan2()` (in module `Base`), 239
`atand()` (in module `Base`), 239

`atanh()` (in module `Base`), 240
`atexit()` (in module `Base`), 211
`available()` (in module `Base.Pkg`), 308
`axpy`
 `()` (in module `Base.LinAlg.BLAS`), 287

B

`backtrace()` (in module `Base`), 221
`baremodule`, 103
`base()` (in module `Base`), 252
`Base.Collections` (module), 233
`Base.Graphics` (module), 309
`Base.LinAlg` (module), 277
`Base.LinAlg.BLAS` (module), 286
`Base.Pkg` (module), 307
`Base.Profile` (module), 316
`Base.Test` (module), 310
`base64()` (in module `Base`), 300
`Base64Pipe()` (in module `Base`), 300
`basename()` (in module `Base`), 292
`beginswith()` (built-in function), 261
`besselh()` (in module `Base`), 245
`besseli()` (in module `Base`), 245
`besselix()` (in module `Base`), 245
`besselj()` (in module `Base`), 245
`besselj0()` (in module `Base`), 245
`besselj1()` (in module `Base`), 245
`besseljx()` (in module `Base`), 245
`besselk()` (in module `Base`), 245
`besselkx()` (in module `Base`), 245
`bessely()` (in module `Base`), 245
`bessely0()` (in module `Base`), 245
`bessely1()` (in module `Base`), 245
`besselyx()` (in module `Base`), 245
`beta()` (in module `Base`), 245
`bfft`
 `()` (in module `Base`), 249
`bfft()` (in module `Base`), 249
`Bidiagonal()` (in module `Base`), 283
`big()` (in module `Base`), 253
`BigFloat()` (in module `Base`), 256
`BigInt()` (in module `Base`), 256
`bin()` (in module `Base`), 252
`bind()` (in module `Base`), 297
`binomial()` (in module `Base`), 243
`bitbroadcast()` (in module `Base`), 266
`bitpack()` (in module `Base`), 271
`bits()` (in module `Base`), 253
`bitunpack()` (in module `Base`), 271
`bkfact`
 `()` (in module `Base`), 280
`bkfact()` (in module `Base`), 280
`blas_set_num_threads()` (in module `Base.LinAlg.BLAS`), 289

blascopy
 () (in module Base.LinAlg.BLAS), 287
 blkdiag() (in module Base), 285
 bool() (in module Base), 253
 BoundingBox() (in module Base.Graphics), 309
 BoundsError (in module Base), 221
 brfft() (in module Base), 250
 broadcast
 () (in module Base), 266
 _function() (in module Base), 266
 broadcast() (in module Base), 266
 broadcast_function() (in module Base), 266
 broadcast_getindex() (in module Base), 266
 broadcast_setindex
 () (in module Base), 267
 bswap() (in module Base), 254
 build() (in module Base.Pkg), 309
 bytes2hex() (in module Base), 255
 bytestring() (built-in function), 259

C

c_malloc() (in module Base), 314
 c_free() (in module Base), 314
 c_malloc() (in module Base), 314
 C_NULL (in module Base), 289
 c_realloc() (in module Base), 314
 cartesianmap() (in module Base), 269
 cat() (in module Base), 267
 catalan (in module Base), 255
 catch_backtrace() (in module Base), 221
 cbrt() (in module Base), 242
 ccall() (in module Base), 313
 Cchar (in module Base), 316
 cd() (in module Base), 290
 Cdouble (in module Base), 316
 ceil() (in module Base), 241
 cell() (in module Base), 265
 center() (in module Base.Graphics), 310
 Cfloat (in module Base), 316
 cfunction() (in module Base), 313
 cglobal() (in module Base), 313
 char() (in module Base), 254
 charwidth() (built-in function), 262
 checkbounds() (in module Base), 268
 checkout() (in module Base.Pkg), 308
 chmod() (in module Base), 290
 chol() (in module Base), 278
 cholfact
 () (in module Base), 279
 cholfact() (in module Base), 279
 chomp() (built-in function), 262
 chop() (built-in function), 262
 chr2ind() (built-in function), 262
 Cint (in module Base), 316
 circshift() (in module Base), 267
 cis() (in module Base), 243
 clamp() (in module Base), 242
 clear() (in module Base.Profile), 317
 clear_malloc_data() (in module Base.Profile), 317
 clipboard() (in module Base), 212
 clone() (in module Base.Pkg), 308
 Clong (in module Base), 316
 Clonglong (in module Base), 316
 close() (in module Base), 294
 cmp() (in module Base), 237
 code_llvm() (in module Base), 223
 code_lowered() (in module Base), 223
 code_native() (in module Base), 223
 code_typed() (in module Base), 223
 Coff_t (in module Base), 316
 collect() (in module Base), 229
 colon() (in module Base), 235
 combinations() (in module Base), 270
 complement
 () (in module Base), 231
 complement() (in module Base), 231
 complex() (in module Base), 254
 complex128() (in module Base), 254
 complex64() (in module Base), 254
 cond() (in module Base), 284
 Condition() (in module Base), 273
 condskeel() (in module Base), 284
 conj
 () (in module Base), 264
 conj() (in module Base), 243
 connect() (in module Base), 297
 consume() (in module Base), 273
 contains() (built-in function), 261
 conv() (in module Base), 251
 conv2() (in module Base), 251
 convert() (in module Base), 214
 copy
 () (in module Base), 315
 copy() (in module Base), 214
 copysign() (in module Base), 242
 cor() (in module Base), 248
 cos() (in module Base), 238
 cosc() (in module Base), 240
 cosd() (in module Base), 238
 cosh() (in module Base), 239
 cospi() (in module Base), 239
 cot() (in module Base), 239
 cotd() (in module Base), 239
 coth() (in module Base), 240
 count() (in module Base), 227
 count_ones() (in module Base), 256
 count_zeros() (in module Base), 256
 countlines() (in module Base), 296

countnz() (in module Base), 264
 cov() (in module Base), 248
 cp() (in module Base), 291
 Cptrdiff_t (in module Base), 316
 CPU_CORES (in module Base), 289
 cross() (in module Base), 277
 csc() (in module Base), 239
 cscd() (in module Base), 239
 csch() (in module Base), 240
 Cshort (in module Base), 316
 Csize_t (in module Base), 316
 Cssize_t (in module Base), 316
 ctime() (in module Base), 290
 ctranspose() (in module Base), 285
 Cuchar (in module Base), 316
 Cuint (in module Base), 316
 Culong (in module Base), 316
 Culonglong (in module Base), 316
 cummax() (in module Base), 268
 cummin() (in module Base), 268
 cumprod
 () (in module Base), 268
 cumprod() (in module Base), 268
 cumsum
 () (in module Base), 268
 cumsum() (in module Base), 268
 cumsum_kbn() (in module Base), 268
 current_module() (in module Base), 222
 current_task() (in module Base), 273
 Cushort (in module Base), 316
 Cwchar_t (in module Base), 316

D

DArray() (in module Base), 276
 dawson() (in module Base), 242
 dct
 () (in module Base), 250
 dct() (in module Base), 250
 dec() (in module Base), 252
 deconv() (in module Base), 251
 deepcopy() (in module Base), 214
 deform() (in module Base.Graphics), 310
 deg2rad() (in module Base), 240
 delete
 () (in module Base), 230
 deleteat
 () (in module Base), 232
 den() (in module Base), 235
 dequeue
 () (in module Base.Collections), 233
 deserialize() (in module Base), 295
 det() (in module Base), 284
 detach() (in module Base), 219
 DevNull (in module Base), 218

dfill() (in module Base), 276
 diag() (in module Base), 283
 diagind() (in module Base), 283
 diagm() (in module Base), 283
 diagonal() (in module Base.Graphics), 310
 Dict() (in module Base), 230
 diff() (in module Base), 268
 digamma() (in module Base), 244
 digits() (in module Base), 253
 dir() (in module Base.Pkg), 307
 dirname() (in module Base), 292
 disable_sigint() (in module Base), 315
 display() (in module Base), 300
 displayable() (in module Base), 301
 distribute() (in module Base), 276
 div() (in module Base), 234
 divrem() (in module Base), 235
 DL_LOAD_PATH (in module Base), 314
 dlclose() (in module Base), 314
 dlopen() (in module Base), 313
 dlopen_e() (in module Base), 314
 dlsym() (in module Base), 314
 dlsym_e() (in module Base), 314
 done() (in module Base), 224
 dones() (in module Base), 276
 dot() (in module Base), 277
 dot() (in module Base.LinAlg.BLAS), 286
 dotc() (in module Base.LinAlg.BLAS), 287
 dotu() (in module Base.LinAlg.BLAS), 287
 download() (in module Base), 291
 drand() (in module Base), 276
 drandn() (in module Base), 276
 dump() (in module Base), 298
 dzeros() (in module Base), 276

E

e (in module Base), 255
 eachline() (in module Base), 299
 eachmatch() (built-in function), 260
 edit() (in module Base), 211
 edit() (in module Base.Pkg), 308
 eig() (in module Base), 280, 281
 eigfact
 () (in module Base), 281
 eigfact() (in module Base), 281
 eigmax() (in module Base), 281
 eigmin() (in module Base), 281
 eigs() (in module Base), 285
 eigvals() (in module Base), 281
 eigvecs() (in module Base), 281
 eltype() (in module Base), 225
 empty
 () (in module Base), 224
 ENDIAN_BOM (in module Base), 295

endof() (in module Base), 224
 endswith() (built-in function), 262
 enqueue
 () (in module Base.Collections), 233
 enumerate() (in module Base), 224
 ENV (in module Base), 220
 EnvHash() (in module Base), 220
 eof() (in module Base), 295
 EOFError (in module Base), 221
 eps() (in module Base), 215
 erf() (in module Base), 242
 erfc() (in module Base), 242
 erfcinv() (in module Base), 242
 erfcx() (in module Base), 242
 erfi() (in module Base), 242
 erfinv() (in module Base), 242
 errno() (in module Base), 315
 error() (in module Base), 221
 ErrorException (in module Base), 221
 esc() (in module Base), 218
 escape_string() (built-in function), 263
 eta() (in module Base), 246
 etree() (in module Base), 272
 eval() (in module Base), 218
 evalfile() (in module Base), 218
 exit() (in module Base), 211
 exp() (in module Base), 241
 exp10() (in module Base), 241
 exp2() (in module Base), 241
 expand() (in module Base), 223
 expanduser() (in module Base), 292
 expm() (in module Base), 285
 expm1() (in module Base), 241
 exponent() (in module Base), 254
 export, 103
 extrema() (in module Base), 226
 eye() (in module Base), 265

F

factor() (in module Base), 243
 factorial() (in module Base), 243
 factorize
 () (in module Base), 278
 factorize() (in module Base), 278
 falses() (in module Base), 265
 fd() (in module Base), 296
 fdio() (in module Base), 294
 fetch() (in module Base), 275
 fetch() (in module Base.Profile), 317
 fft
 () (in module Base), 248
 fft() (in module Base), 248
 fftshift() (in module Base), 250
 fieldoffsets() (in module Base), 216

fieldtype() (in module Base), 216
 filemode() (in module Base), 290
 filesize() (in module Base), 290
 fill
 () (in module Base), 265
 fill() (in module Base), 265
 filt
 () (in module Base), 251
 filt() (in module Base), 251
 filter
 () (in module Base), 229
 filter() (in module Base), 229
 finalizer() (in module Base), 214
 find() (in module Base), 267
 find_library() (in module Base), 314
 findfirst() (in module Base), 267
 findin() (in module Base), 225
 findmax() (in module Base), 226
 findmin() (in module Base), 226
 findn() (in module Base), 267
 findnext() (in module Base), 267, 268
 findnz() (in module Base), 267
 first() (in module Base), 228
 fld() (in module Base), 235
 flipbits
 () (in module Base), 271
 flipdim() (in module Base), 267
 fliplr() (in module Base), 267
 flipsign() (in module Base), 242
 flipud() (in module Base), 267
 float() (in module Base), 254
 float16() (in module Base), 254
 float32() (in module Base), 254
 float32_isvalid() (in module Base), 254
 float64() (in module Base), 254
 float64_isvalid() (in module Base), 254
 floor() (in module Base), 241
 flush() (in module Base), 294
 flush_cstdio() (in module Base), 294
 foldl() (in module Base), 225
 foldr() (in module Base), 225, 226
 free() (in module Base.Pkg), 309
 frexp() (in module Base), 241
 full() (in module Base), 271
 fullname() (in module Base), 222
 function_module() (in module Base), 222
 function_name() (in module Base), 222
 functionloc() (in module Base), 222
 functionlocs() (in module Base), 222

G

gamma() (in module Base), 244
 gbm
 () (in module Base.LinAlg.BLAS), 287

gbmv() (in module Base.LinAlg.BLAS), 287
gc() (in module Base), 222
gc_disable() (in module Base), 223
gc_enable() (in module Base), 223
gcd() (in module Base), 243
gcdx() (in module Base), 243
gemm
 () (in module Base.LinAlg.BLAS), 288
gemm() (in module Base.LinAlg.BLAS), 288
gemv
 () (in module Base.LinAlg.BLAS), 288
gemv() (in module Base.LinAlg.BLAS), 288
generate() (in module Base.Pkg), 309
gensym() (in module Base), 218
get
 () (in module Base), 230
get() (in module Base), 230
get_bigfloat_precision() (in module Base), 258
get_rounding() (in module Base), 256
getaddrinfo() (in module Base), 297
getfield() (in module Base), 216
gethostname() (in module Base), 219
getindex() (in module Base), 229, 265, 266
getipaddr() (in module Base), 219
getkey() (in module Base), 230
getpid() (in module Base), 219
gperm() (in module Base), 291
gradient() (in module Base), 268

H

hankelh1() (in module Base), 245
hankelh1x() (in module Base), 245
hankelh2() (in module Base), 245
hankelh2x() (in module Base), 245
hash() (in module Base), 214
haskey() (in module Base), 230
hcat() (in module Base), 267
heapify
 () (in module Base.Collections), 234
heapify() (in module Base.Collections), 233
heappop
 () (in module Base.Collections), 234
heappush
 () (in module Base.Collections), 234
height() (in module Base.Graphics), 310
help() (in module Base), 212
herk
 () (in module Base.LinAlg.BLAS), 287
herk() (in module Base.LinAlg.BLAS), 287
hessfact
 () (in module Base), 282
hessfact() (in module Base), 281
hex() (in module Base), 252
hex2bytes() (in module Base), 255

hex2num() (in module Base), 254
hist
 () (in module Base), 247
hist() (in module Base), 247
hist2d
 () (in module Base), 247
hist2d() (in module Base), 247
histrange() (in module Base), 247
homedir() (in module Base), 292
htol() (in module Base), 295
hton() (in module Base), 295
hvcat() (in module Base), 267
hypot() (in module Base), 240
|
iceil() (in module Base), 241
idct
 () (in module Base), 250
idct() (in module Base), 250
identity() (in module Base), 215
ifelse() (in module Base), 213
ifft
 () (in module Base), 249
ifft() (in module Base), 248
ifftshift() (in module Base), 251
ifloor() (in module Base), 241
ignorestatus() (in module Base), 219
im (in module Base), 255
imag() (in module Base), 242
import, 103
importall, 103
in() (in module Base), 225
include() (in module Base), 212
include_string() (in module Base), 212
ind2chr() (built-in function), 262
ind2sub() (in module Base), 264
indexin() (in module Base), 225
indexpids() (in module Base), 277
indmax() (in module Base), 226
indmin() (in module Base), 226
Inf (in module Base), 255
inf() (in module Base), 256
Inf16 (in module Base), 255
Inf32 (in module Base), 255
info() (in module Base), 298
init() (in module Base.Pkg), 307
init() (in module Base.Profile), 317
insert
 () (in module Base), 232
installed() (in module Base.Pkg), 308
int() (in module Base), 253
int128() (in module Base), 253
int16() (in module Base), 253
int32() (in module Base), 253

int64() (in module Base), 253
 int8() (in module Base), 253
 integer() (in module Base), 253
 interrupt() (in module Base), 274
 intersect
 () (in module Base), 231
 intersect() (in module Base), 231
 IntSet() (in module Base), 231
 inv() (in module Base), 284
 invdigamma() (in module Base), 244
 invmod() (in module Base), 244
 invoke() (in module Base), 217
 invperm() (in module Base), 270
 IOBuffer() (in module Base), 293
 ipermute
 () (in module Base), 270
 ipermutedims() (in module Base), 268
 IPv4() (in module Base), 297
 IPv6() (in module Base), 297
 irfft() (in module Base), 250
 iround() (in module Base), 241
 is() (in module Base), 213
 is_assigned_char() (built-in function), 260
 is_valid_ascii() (built-in function), 260
 is_valid_char() (built-in function), 260
 is_valid_utf16() (built-in function), 263
 is_valid_utf8() (built-in function), 260
 isa() (in module Base), 213
 isabspath() (in module Base), 292
 isalnum() (built-in function), 262
 isalpha() (built-in function), 262
 isapprox() (in module Base), 238
 isascii() (built-in function), 262
 isbits() (in module Base), 216
 isblank() (built-in function), 263
 isblockdev() (in module Base), 291
 ischardev() (in module Base), 291
 iscntrl() (built-in function), 263
 isconst() (in module Base), 222
 isdefined() (in module Base), 214
 isdigit() (built-in function), 263
 isdir() (in module Base), 291
 isdirpath() (in module Base), 292
 iseltype() (in module Base), 264
 isempty() (in module Base), 224
 isequal() (in module Base), 213
 iseven() (in module Base), 257
 isexecutable() (in module Base), 291
 isfifo() (in module Base), 291
 isfile() (in module Base), 292
 isfinite() (in module Base), 255
 isgeneric() (in module Base), 222
 isgraph() (built-in function), 263
 isheap() (in module Base.Collections), 234
 ishermitian() (in module Base), 285
 isimmutable() (in module Base), 216
 isinf() (in module Base), 255
 isinside() (in module Base.Graphics), 310
 isinteger() (in module Base), 256
 isinteractive() (in module Base), 211
 isleaftype() (in module Base), 216
 isless() (in module Base), 213
 islink() (in module Base), 292
 islower() (built-in function), 263
 ismarked() (in module Base), 295
 ismatch() (built-in function), 260
 isnan() (in module Base), 256
 isodd() (in module Base), 257
 isopen() (in module Base), 295
 ispath() (in module Base), 292
 isperm() (in module Base), 270
 isposdef
 () (in module Base), 285
 isposdef() (in module Base), 285
 ispow2() (in module Base), 243
 isprime() (in module Base), 257
 isprint() (built-in function), 263
 ispunct() (built-in function), 263
 isqrt() (in module Base), 242
 isreadable() (in module Base), 292
 isreadonly() (in module Base), 295
 isready() (in module Base), 275
 isreal() (in module Base), 256
 issetgid() (in module Base), 292
 issetuid() (in module Base), 292
 issocket() (in module Base), 292
 issorted() (in module Base), 306
 isspace() (built-in function), 263
 issparse() (in module Base), 271
 issticky() (in module Base), 292
 issubnormal() (in module Base), 255
 issubset() (in module Base), 229, 232
 issubtype() (in module Base), 215
 issym() (in module Base), 285
 istaskdone() (in module Base), 273
 istext() (in module Base), 302
 istril() (in module Base), 285
 istriu() (in module Base), 285
 isupper() (built-in function), 263
 isvalid() (built-in function), 262
 iswritable() (in module Base), 292
 isxdigit() (built-in function), 263
 itrunc() (in module Base), 241

J

join() (built-in function), 262
 joinpath() (in module Base), 292

K

`KeyError` (in module `Base`), 221
`keys()` (in module `Base`), 230
`kill()` (in module `Base`), 219
`kron()` (in module `Base`), 284

L

`last()` (in module `Base`), 228
`lbeta()` (in module `Base`), 245
`lcfirsr()` (built-in function), 262
`lcm()` (in module `Base`), 243
`ldexp()` (in module `Base`), 241
`ldlrfact()` (in module `Base`), 279
`leading_ones()` (in module `Base`), 257
`leading_zeros()` (in module `Base`), 257
`length()` (built-in function), 259
`length()` (in module `Base`), 224, 264
`less()` (in module `Base`), 212
`lexcmp()` (in module `Base`), 213
`lexless()` (in module `Base`), 214
`lfact()` (in module `Base`), 244
`lgamma()` (in module `Base`), 244
`linrange()` (in module `Base`), 235
`linreg()` (in module `Base`), 285
`linspace()` (in module `Base`), 266
`listen()` (in module `Base`), 297
`listenany()` (in module `Base`), 297
`LOAD_PATH` (in module `Base`), 289
`LoadError` (in module `Base`), 221
`localindexes()` (in module `Base`), 277
`localpart()` (in module `Base`), 277
`log()` (in module `Base`), 240
`log10()` (in module `Base`), 240
`log1p()` (in module `Base`), 241
`log2()` (in module `Base`), 240
`logdet()` (in module `Base`), 284
`logspace()` (in module `Base`), 266
`lowercase()` (built-in function), 262
`lpad()` (built-in function), 260
`lstat()` (in module `Base`), 290
`lstrip()` (built-in function), 261
`ltoh()` (in module `Base`), 295
`lu()` (in module `Base`), 278
`lufact`
 `()` (in module `Base`), 278
`lufact()` (in module `Base`), 278
`lyap()` (in module `Base`), 285

M

`macroexpand()` (in module `Base`), 223
`map`
 `()` (in module `Base`), 228
`map()` (in module `Base`), 228

`mapfoldl()` (in module `Base`), 228
`mapfoldr()` (in module `Base`), 228
`mapreduce()` (in module `Base`), 228
`mapslices()` (in module `Base`), 269
`mark()` (in module `Base`), 294
`match()` (built-in function), 260
`matchall()` (built-in function), 260
`max()` (in module `Base`), 241
`maxabs`
 `()` (in module `Base`), 226
`maxabs()` (in module `Base`), 226
`maximum`
 `()` (in module `Base`), 226
`maximum()` (in module `Base`), 226
`maxintfloat()` (in module `Base`), 215
`mean`
 `()` (in module `Base`), 246
`mean()` (in module `Base`), 246
`median`
 `()` (in module `Base`), 247
`median()` (in module `Base`), 247
`merge`
 `()` (in module `Base`), 230
`merge()` (in module `Base`), 230
`MersenneTwister()` (in module `Base`), 258
`method_exists()` (in module `Base`), 217
`MethodError` (in module `Base`), 221
`methods()` (in module `Base`), 212
`methodswith()` (in module `Base`), 213
`middle()` (in module `Base`), 246, 247
`midpoints()` (in module `Base`), 247
`mimewritable()` (in module `Base`), 301
`min()` (in module `Base`), 241
`minabs`
 `()` (in module `Base`), 226
`minabs()` (in module `Base`), 226
`minimum`
 `()` (in module `Base`), 226
`minimum()` (in module `Base`), 226
`minmax()` (in module `Base`), 242
`mkdir()` (in module `Base`), 290
`mkpath()` (in module `Base`), 290
`mktemp()` (in module `Base`), 291
`mktempdir()` (in module `Base`), 291
`mmap()` (in module `Base`), 303
`mmap_array()` (in module `Base`), 302
`mmap_bitarray()` (in module `Base`), 303
`mod()` (in module `Base`), 235
`mod1()` (in module `Base`), 235
`mod2pi()` (in module `Base`), 235
`modf()` (in module `Base`), 241
`module`, 103
`module_name()` (in module `Base`), 222
`module_parent()` (in module `Base`), 222

MS_ASYNC (in module Base), 303
 MS_INVALIDATE (in module Base), 303
 MS_SYNC (in module Base), 303
 msync() (in module Base), 303
 mtime() (in module Base), 290
 munmap() (in module Base), 303
 mv() (in module Base), 291
 myid() (in module Base), 274

N

names() (in module Base), 222
 NaN (in module Base), 255
 nan() (in module Base), 256
 NaN16 (in module Base), 255
 NaN32 (in module Base), 255
 nb_available() (in module Base), 297
 ndigits() (in module Base), 246
 ndims() (in module Base), 264
 next() (in module Base), 224
 nextfloat() (in module Base), 256
 nextind() (built-in function), 262
 nextpow() (in module Base), 244
 nextpow2() (in module Base), 243
 nextprod() (in module Base), 244
 nnz() (in module Base), 271
 nonzeros() (in module Base), 272
 norm() (in module Base), 284
 normalize_string() (built-in function), 259
 normpath() (in module Base), 292
 notify() (in module Base), 273
 nprocs() (in module Base), 274
 nrm2() (in module Base.LinAlg.BLAS), 287
 nthperm
 () (in module Base), 269
 nthperm() (in module Base), 269
 ntoh() (in module Base), 295
 ntuple() (in module Base), 214
 null() (in module Base), 284
 num() (in module Base), 235
 num2hex() (in module Base), 254
 nworkers() (in module Base), 274

O

object_id() (in module Base), 214
 oct() (in module Base), 252
 oftype() (in module Base), 214
 one() (in module Base), 255
 ones() (in module Base), 265
 open() (in module Base), 219, 293
 operm() (in module Base), 291
 OS_NAME (in module Base), 289

P

parent() (in module Base), 266

parentindexes() (in module Base), 266
 parse() (in module Base), 218
 ParseError (in module Base), 221
 parsefloat() (in module Base), 253
 parseint() (in module Base), 253
 parseip() (in module Base), 297
 partitions() (in module Base), 270
 peakflops() (in module Base), 286
 peek() (in module Base.Collections), 233
 permutations() (in module Base), 270
 permute
 () (in module Base), 270
 permutedims() (in module Base), 268
 pi (in module Base), 255
 pin() (in module Base.Pkg), 308
 pinv() (in module Base), 284
 PipeBuffer() (in module Base), 296
 plan_bfft
 () (in module Base), 249
 plan_bfft() (in module Base), 249
 plan_brfft() (in module Base), 250
 plan_dct
 () (in module Base), 250
 plan_dct() (in module Base), 250
 plan_fft
 () (in module Base), 249
 plan_fft() (in module Base), 249
 plan_idct
 () (in module Base), 250
 plan_idct() (in module Base), 250
 plan_ift
 () (in module Base), 249
 plan_ift() (in module Base), 249
 plan_irfft() (in module Base), 250
 plan_r2r
 () (in module Base.FFTW), 251
 plan_r2r() (in module Base.FFTW), 251
 plan_rfft() (in module Base), 250
 pmap() (in module Base), 274
 pointer() (in module Base), 315
 pointer_from_objref() (in module Base), 315
 pointer_to_array() (in module Base), 315
 poll_fd() (in module Base), 297
 poll_file() (in module Base), 297
 polygamma() (in module Base), 244
 pop
 () (in module Base), 230, 232
 popdisplay() (in module Base), 302
 position() (in module Base), 294
 powermod() (in module Base), 244
 precision() (in module Base), 257
 precompile() (in module Base), 223
 prepend
 () (in module Base), 233

[prevfloat\(\)](#) (in module Base), 256
[prevind\(\)](#) (built-in function), 262
[prevpow\(\)](#) (in module Base), 244
[prevpow2\(\)](#) (in module Base), 244
[prevprod\(\)](#) (in module Base), 244
[primes\(\)](#) (in module Base), 257
[print\(\)](#) (in module Base), 298
[print\(\)](#) (in module Base.Profile), 317
[print_escaped\(\)](#) (in module Base), 295
[print_joined\(\)](#) (in module Base), 295
[print_shortest\(\)](#) (in module Base), 296
[print_unescaped\(\)](#) (in module Base), 295
[print_with_color\(\)](#) (in module Base), 298
[println\(\)](#) (in module Base), 298
[PriorityQueue{K,V}\(\)](#) (in module Base.Collections), 233
[process_exited\(\)](#) (in module Base), 218
[process_running\(\)](#) (in module Base), 218
[ProcessExitedException](#) (in module Base), 221
[procs\(\)](#) (in module Base), 274, 277
[prod](#)
 [\(\)](#) (in module Base), 227
[prod\(\)](#) (in module Base), 227
[produce\(\)](#) (in module Base), 273
[promote\(\)](#) (in module Base), 214
[promote_rule\(\)](#) (in module Base), 216
[promote_shape\(\)](#) (in module Base), 268
[promote_type\(\)](#) (in module Base), 216
[publish\(\)](#) (in module Base.Pkg), 309
[push](#)
 [\(\)](#) (in module Base), 232
[pushdisplay\(\)](#) (in module Base), 302
[put](#)
 [\(\)](#) (in module Base), 275
[pwd\(\)](#) (in module Base), 290

Q

[qr\(\)](#) (in module Base), 279
[qrfact](#)
 [\(\)](#) (in module Base), 280
[qrfact\(\)](#) (in module Base), 279
[quadgk\(\)](#) (in module Base), 251
[quantile](#)
 [\(\)](#) (in module Base), 247
[quantile\(\)](#) (in module Base), 247
[quit\(\)](#) (in module Base), 211

R

[r2r](#)
 [\(\)](#) (in module Base.FFTW), 251
[r2r\(\)](#) (in module Base.FFTW), 251
[rad2deg\(\)](#) (in module Base), 240
[rand](#)
 [\(\)](#) (in module Base), 258
[rand\(\)](#) (in module Base), 258

[randbool](#)
 [\(\)](#) (in module Base), 258
[randbool\(\)](#) (in module Base), 258
[randcycle\(\)](#) (in module Base), 270
[randn](#)
 [\(\)](#) (in module Base), 259
[randn\(\)](#) (in module Base), 258
[randperm\(\)](#) (in module Base), 270
[randstring\(\)](#) (built-in function), 262
[randsubseq](#)
 [\(\)](#) (in module Base), 268
[randsubseq\(\)](#) (in module Base), 268
[range\(\)](#) (in module Base), 235
[rank\(\)](#) (in module Base), 284
[rationalize\(\)](#) (in module Base), 235
[read](#)
 [\(\)](#) (in module Base), 294
[read\(\)](#) (in module Base), 294
[readall\(\)](#) (in module Base), 299
[readandwrite\(\)](#) (in module Base), 219
[readavailable\(\)](#) (in module Base), 296
[readbytes](#)
 [\(\)](#) (in module Base), 294
[readbytes\(\)](#) (in module Base), 294
[readchomp\(\)](#) (in module Base), 296
[readcsv\(\)](#) (in module Base), 300
[readdir\(\)](#) (in module Base), 296
[readdlm\(\)](#) (in module Base), 299, 300
[readline\(\)](#) (in module Base), 299
[readlines\(\)](#) (in module Base), 299
[readuntil\(\)](#) (in module Base), 299
[real\(\)](#) (in module Base), 242
[realmax\(\)](#) (in module Base), 215
[realmin\(\)](#) (in module Base), 215
[realpath\(\)](#) (in module Base), 292
[recv\(\)](#) (in module Base), 297
[redirect_stderr\(\)](#) (in module Base), 296
[redirect_stdin\(\)](#) (in module Base), 296
[redirect_stdout\(\)](#) (in module Base), 296
[redisplay\(\)](#) (in module Base), 301
[reduce\(\)](#) (in module Base), 225
[reducedim\(\)](#) (in module Base), 269
[reenable_sigint\(\)](#) (in module Base), 315
[register\(\)](#) (in module Base.Pkg), 309
[reim\(\)](#) (in module Base), 243
[reinterpret\(\)](#) (in module Base), 265
[reload\(\)](#) (in module Base), 212
[rem\(\)](#) (in module Base), 235
[rem1\(\)](#) (in module Base), 235
[remotecall\(\)](#) (in module Base), 274
[remotecall_fetch\(\)](#) (in module Base), 275
[remotecall_wait\(\)](#) (in module Base), 275
[RemoteRef\(\)](#) (in module Base), 275
[repeat\(\)](#) (in module Base), 284

- replace() (built-in function), 261
 - repmat() (in module Base), 284
 - repr() (built-in function), 259
 - reprmime() (in module Base), 301
 - require() (in module Base), 212
 - reset() (in module Base), 295
 - reshape() (in module Base), 265
 - resize
 - () (in module Base), 232
 - resolve() (in module Base.Pkg), 308
 - rethrow() (in module Base), 221
 - retrieve() (in module Base.Profile), 317
 - reverse
 - () (in module Base), 270
 - reverse() (in module Base), 270
 - rfft() (in module Base), 249
 - rm() (in module Base), 291
 - rm() (in module Base.Pkg), 308
 - rmprocs() (in module Base), 274
 - rol() (in module Base), 271
 - ror() (in module Base), 271
 - rot180() (in module Base), 269
 - rotate() (in module Base.Graphics), 310
 - rotl90() (in module Base), 269
 - rotr90() (in module Base), 269
 - round() (in module Base), 241
 - rpad() (built-in function), 261
 - rref() (in module Base), 278
 - rsearch() (built-in function), 261
 - rsearchindex() (built-in function), 261
 - rsplit() (built-in function), 261
 - rstrip() (built-in function), 261
 - RTLD_DEEPBIND (in module Base), 314
 - RTLD_FIRST (in module Base), 314
 - RTLD_GLOBAL (in module Base), 314
 - RTLD_LAZY (in module Base), 314
 - RTLD_LOCAL (in module Base), 314
 - RTLD_NODELETE (in module Base), 314
 - RTLD_NOLOAD (in module Base), 314
 - RTLD_NOW (in module Base), 314
 - run() (in module Base), 218
 - runttests() (in module Base), 313
- ## S
- sbmv
 - () (in module Base.LinAlg.BLAS), 287
 - sbmv() (in module Base.LinAlg.BLAS), 287, 288
 - scal
 - () (in module Base.LinAlg.BLAS), 287
 - scal() (in module Base.LinAlg.BLAS), 287
 - scale
 - () (in module Base), 283
 - scale() (in module Base), 283
 - schedule() (in module Base), 273
 - schur() (in module Base), 282
 - schurfact
 - () (in module Base), 282
 - schurfact() (in module Base), 282
 - sdata() (in module Base), 277
 - search() (built-in function), 261
 - searchindex() (built-in function), 261
 - searchsorted() (in module Base), 306
 - searchsortedfirst() (in module Base), 306
 - searchsortedlast() (in module Base), 307
 - sec() (in module Base), 239
 - secd() (in module Base), 239
 - sech() (in module Base), 240
 - seek() (in module Base), 294
 - seekend() (in module Base), 294
 - seekstart() (in module Base), 294
 - select
 - () (in module Base), 307
 - select() (in module Base), 307
 - send() (in module Base), 297
 - serialize() (in module Base), 295
 - Set() (in module Base), 231
 - set_bigfloat_precision() (in module Base), 258
 - set_rounding() (in module Base), 256
 - setdiff
 - () (in module Base), 231
 - setdiff() (in module Base), 231
 - setenv() (in module Base), 219
 - setfield
 - () (in module Base), 216
 - setindex
 - () (in module Base), 229, 266
 - setopt() (in module Base), 298
 - SharedArray() (in module Base), 277
 - shift
 - () (in module Base), 232
 - shift() (in module Base.Graphics), 310
 - show() (in module Base), 298
 - showall() (in module Base), 298
 - showcompact() (in module Base), 298
 - showerror() (in module Base), 298
 - shuffle
 - () (in module Base), 270
 - shuffle() (in module Base), 270
 - sign() (in module Base), 242
 - signbit() (in module Base), 242
 - signed() (in module Base), 253
 - signif() (in module Base), 241
 - significand() (in module Base), 254
 - similar() (in module Base), 265
 - sin() (in module Base), 238
 - sinc() (in module Base), 240
 - sind() (in module Base), 238
 - sinh() (in module Base), 239

`sinpi()` (in module Base), 239
`size()` (in module Base), 264
`sizehint()` (in module Base), 231
`sizeof()` (built-in function), 259
`sizeof()` (in module Base), 215
`skip()` (in module Base), 294
`skipchars()` (in module Base), 296
`sleep()` (in module Base), 273
`slice()` (in module Base), 266
`slicedim()` (in module Base), 266
`sort`
 `()` (in module Base), 306
`sort()` (in module Base), 306
`sortcols()` (in module Base), 306
`sortperm()` (in module Base), 306
`sortrows()` (in module Base), 306
`sparse()` (in module Base), 271
`sparsevec()` (in module Base), 271
`spawn()` (in module Base), 218
`spdiagm()` (in module Base), 272
`speye()` (in module Base), 272
`splice`
 `()` (in module Base), 232
`split()` (built-in function), 261
`splitdir()` (in module Base), 292
`splitdrive()` (in module Base), 293
`splitext()` (in module Base), 293
`spones()` (in module Base), 272
`sprand()` (in module Base), 272
`sprandbool()` (in module Base), 272
`sprandn()` (in module Base), 272
`sprint()` (in module Base), 298
`spzeros()` (in module Base), 272
`sqrt()` (in module Base), 242
`sqrtm()` (in module Base), 280
`squeeze()` (in module Base), 268
`rand()` (in module Base), 258
`start()` (in module Base), 224
`start_timer()` (in module Base), 222
`stat()` (in module Base), 290
`status()` (in module Base.Pkg), 308
`std()` (in module Base), 246
`STDERR` (in module Base), 293
`STDIN` (in module Base), 293
`stdm()` (in module Base), 246
`STDOUT` (in module Base), 293
`step()` (in module Base), 229
`stop_timer()` (in module Base), 222
`strerror()` (in module Base), 316
`strftime()` (in module Base), 220
`stride()` (in module Base), 264
`strides()` (in module Base), 264
`string()` (built-in function), 259
`stringmime()` (in module Base), 301

`strip()` (built-in function), 261
`strptime()` (in module Base), 220
`strwidth()` (built-in function), 262
`sub()` (in module Base), 266
`sub2ind()` (in module Base), 264
`subtypes()` (in module Base), 215
`subtypetree()` (in module Base), 215
`success()` (in module Base), 218
`sum`
 `()` (in module Base), 227
`sum()` (in module Base), 226, 227
`sum_kbn()` (in module Base), 269
`sumabs`
 `()` (in module Base), 227
`sumabs()` (in module Base), 227
`sumabs2`
 `()` (in module Base), 227
`sumabs2()` (in module Base), 227
`summary()` (in module Base), 298
`super()` (in module Base), 215
`svd()` (in module Base), 282
`svdfact`
 `()` (in module Base), 282
`svdfact()` (in module Base), 282
`svdvals`
 `()` (in module Base), 282
`svdvals()` (in module Base), 282, 283
`sylvester()` (in module Base), 285
`symbol()` (built-in function), 263
`syndiff`
 `()` (in module Base), 231
`syndiff()` (in module Base), 231
`symlink()` (in module Base), 290
`symm`
 `()` (in module Base.LinAlg.BLAS), 288
`symm()` (in module Base.LinAlg.BLAS), 288
`symperm()` (in module Base), 272
`SymTridiagonal()` (in module Base), 283
`symv`
 `()` (in module Base.LinAlg.BLAS), 288
`symv()` (in module Base.LinAlg.BLAS), 288
`syrk`
 `()` (in module Base.LinAlg.BLAS), 287
`syrk()` (in module Base.LinAlg.BLAS), 287
`SystemError` (in module Base), 221
`systemerror()` (in module Base), 316

T

`tag()` (in module Base.Pkg), 309
`take`
 `()` (in module Base), 275
`takebuf_array()` (in module Base), 294
`takebuf_string()` (in module Base), 294
`tan()` (in module Base), 238

tand() (in module Base), 239
 tanh() (in module Base), 239
 Task() (in module Base), 272
 task_local_storage() (in module Base), 273
 tempdir() (in module Base), 291
 tempname() (in module Base), 291
 test() (in module Base.Pkg), 309
 TextDisplay() (in module Base), 302
 throw() (in module Base), 221
 tic() (in module Base), 220
 time() (in module Base), 219
 time_ns() (in module Base), 220
 timedwait() (in module Base), 275
 Timer() (in module Base), 222
 TmStruct() (in module Base), 220
 toc() (in module Base), 220
 toq() (in module Base), 220
 touch() (in module Base), 291
 trace() (in module Base), 284
 trailing_ones() (in module Base), 257
 trailing_zeros() (in module Base), 257
 transpose() (in module Base), 285
 Tridiagonal() (in module Base), 283
 trigamma() (in module Base), 244
 tril
 () (in module Base), 283
 tril() (in module Base), 283
 triu
 () (in module Base), 283
 triu() (in module Base), 283
 trmm
 () (in module Base.LinAlg.BLAS), 288
 trmm() (in module Base.LinAlg.BLAS), 288
 trmv
 () (in module Base.LinAlg.BLAS), 289
 trmv() (in module Base.LinAlg.BLAS), 289
 trsm
 () (in module Base.LinAlg.BLAS), 288
 trsm() (in module Base.LinAlg.BLAS), 289
 trsv
 () (in module Base.LinAlg.BLAS), 289
 trsv() (in module Base.LinAlg.BLAS), 289
 trues() (in module Base), 265
 trunc() (in module Base), 241
 truncate() (in module Base), 296
 tuple() (in module Base), 214
 TypeError (in module Base), 221
 typeintersect() (in module Base), 217
 typejoin() (in module Base), 216
 typemax() (in module Base), 215
 typemin() (in module Base), 215
 typeof() (in module Base), 214

U

ucfirst() (built-in function), 262
 uint() (in module Base), 253
 uint128() (in module Base), 254
 uint16() (in module Base), 253
 uint32() (in module Base), 253
 uint64() (in module Base), 254
 uint8() (in module Base), 253
 unescape_string() (built-in function), 263
 union
 () (in module Base), 231
 union() (in module Base), 231
 unique() (in module Base), 225
 unmark() (in module Base), 294
 unsafe_copy
 () (in module Base), 315
 unsafe_load() (in module Base), 314
 unsafe_pointer_to_objref() (in module Base), 315
 unsafe_store
 () (in module Base), 315
 unshift
 () (in module Base), 232
 unsigned() (in module Base), 253
 update() (in module Base.Pkg), 308
 uperm() (in module Base), 291
 uppercase() (built-in function), 262
 using, 103
 utf16() (built-in function), 263
 utf32() (built-in function), 264
 utf8() (built-in function), 259

V

values() (in module Base), 230
 var() (in module Base), 246
 varm() (in module Base), 246
 vcat() (in module Base), 267
 vec() (in module Base), 268
 Vec2() (in module Base.Graphics), 309
 vecnorm() (in module Base), 284
 VERSION (in module Base), 289
 versioninfo() (in module Base), 213

W

wait() (in module Base), 274
 warn() (in module Base), 298
 watch_file() (in module Base), 297
 which() (in module Base), 212
 whos() (in module Base), 211
 widemul() (in module Base), 246
 widen() (in module Base), 215
 width() (in module Base.Graphics), 310
 with_bigfloat_precision() (in module Base), 258
 with_handler() (in module Base.Test), 313

`with_rounding()` (in module Base), 256
`Woodbury()` (in module Base), 283
`WORD_SIZE` (in module Base), 289
`workers()` (in module Base), 274
`workspace()` (in module Base), 213
`write()` (in module Base), 294
`writcsv()` (in module Base), 300
`writedlm()` (in module Base), 300
`writemime()` (in module Base), 301
`wstring()` (built-in function), 264

X

`xcorr()` (in module Base), 251
`xdump()` (in module Base), 299
`xmax()` (in module Base.Graphics), 310
`xmin()` (in module Base.Graphics), 310
`xrange()` (in module Base.Graphics), 310

Y

`yield()` (in module Base), 273
`yieldto()` (in module Base), 272
`ymax()` (in module Base.Graphics), 310
`ymin()` (in module Base.Graphics), 310
`yrange()` (in module Base.Graphics), 310

Z

`zero()` (in module Base), 255
`zeros()` (in module Base), 265
`zeta()` (in module Base), 246
`zip()` (in module Base), 224