# Contents

1 The Julia Manual

1.1 Introduction .................................................. 1
1.2 Getting Started ............................................ 2
1.3 Variables ...................................................... 4
1.4 Integers and Floating-Point Numbers ..................... 5
1.5 Mathematical Operations and Elementary Functions .... 15
1.6 Complex and Rational Numbers ........................... 22
1.7 Strings ......................................................... 26
1.8 Functions ....................................................... 36
1.9 Control Flow .................................................. 44
1.10 Scope of Variables .......................................... 55
1.11 Types ........................................................ 60
1.12 Methods ......................................................... 74
1.13 Constructors ................................................ 81
1.14 Conversion and Promotion .................................. 88
1.15 Modules ......................................................... 93
1.16 Metaprogramming ........................................... 95
1.17 Multi-dimensional Arrays .................................. 103
1.18 Linear algebra ................................................ 112
1.19 Networking and Streams ................................... 114
1.20 Parallel Computing .......................................... 117
1.21 Running External Programs ................................. 125
1.22 Calling C and Fortran Code ................................. 129
1.23 Packages ...................................................... 136
1.24 Performance Tips ............................................ 145
1.25 Style Guide .................................................... 149
1.26 Frequently Asked Questions ................................. 153
1.27 Noteworthy Differences from other Languages ......... 160

2 The Julia Standard Library ........................................ 163

2.1 Built-ins ......................................................... 163
2.2 Built-in Modules .............................................. 245

Bibliography .......................................................... 257
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.

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 transcends its general programming limitations. 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.
| | | | | | /_/ |
| | |_| | | | (_| | | Commit 61b47c5aa7 (2011-08-20 06:11:31)*
| \__/ | \__/ | \__/ |

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 \texttt{ans}
is bound to the value of the last evaluated expression whether it is shown or not. The \texttt{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 \texttt{file.jl}, write \texttt{include("file.jl")}.

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

\texttt{$ julia script.jl arg1 arg2...}$

As the example implies, the following command-line arguments to \texttt{julia} are taken as command-line arguments to the
program \texttt{script.jl}, passed in the global constant \texttt{ARGS}. \texttt{ARGS} is also set when script code is given using the \texttt{-e}
option on the command line (see the \texttt{julia} help output below). For example, to just print the arguments given to a
script, you could do this:

\texttt{$ julia -e 'for x in ARGS; println(x); end' foo bar}$

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

\texttt{$ echo 'for x in ARGS; println(x); end' > script.jl}$

\texttt{$ julia script.jl foo bar}$

\texttt{foo}

\texttt{bar}

Julia can be started in parallel mode with either the \texttt{-p} or the \texttt{--machinefile} options. \texttt{-p n} will launch
an additional \texttt{n} worker processes, while \texttt{--machinefile file} will launch a worker for each line in file
\texttt{file}. The machines defined in \texttt{file} must be accessible via \texttt{ssh} and each machine definition takes the form
\texttt{[user@]host[:port]}

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

\texttt{$ echo 'println("Greetings! 好！안녕하세요?");' > ~/.juliarc.jl}$

\texttt{$ julia}$

Greetings! 好！안녕하세요?

...
### 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

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> ix = 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
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
0.00001

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

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
```

julia> pi
3

julia> sqrt = 4
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 Unicode character with code point greater than 00A0. Subsequent characters may also include ! and digits (0-9).

All operators 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.

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 CamelCase instead of underscores.
- Names of functions and macros are in lowercase, 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Signed?</th>
<th>Number of bits</th>
<th>Smallest value</th>
<th>Largest value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8</td>
<td>x</td>
<td>8</td>
<td>-2^7</td>
<td>2^7 - 1</td>
</tr>
<tr>
<td>Uint8</td>
<td></td>
<td>8</td>
<td>0</td>
<td>2^8 - 1</td>
</tr>
<tr>
<td>Int16</td>
<td>x</td>
<td>16</td>
<td>-2^15</td>
<td>2^15 - 1</td>
</tr>
<tr>
<td>Uint16</td>
<td></td>
<td>16</td>
<td>0</td>
<td>2^16 - 1</td>
</tr>
<tr>
<td>Int32</td>
<td>x</td>
<td>32</td>
<td>-2^31</td>
<td>2^31 - 1</td>
</tr>
<tr>
<td>Uint32</td>
<td></td>
<td>32</td>
<td>0</td>
<td>2^32 - 1</td>
</tr>
<tr>
<td>Int64</td>
<td>x</td>
<td>64</td>
<td>-2^63</td>
<td>2^63 - 1</td>
</tr>
<tr>
<td>Uint64</td>
<td></td>
<td>64</td>
<td>0</td>
<td>2^64 - 1</td>
</tr>
<tr>
<td>Int128</td>
<td>x</td>
<td>128</td>
<td>-2^127</td>
<td>2^127 - 1</td>
</tr>
<tr>
<td>Uint128</td>
<td></td>
<td>128</td>
<td>0</td>
<td>2^128 - 1</td>
</tr>
<tr>
<td>Bool</td>
<td>N/A</td>
<td>8</td>
<td>false (0)</td>
<td>true (1)</td>
</tr>
<tr>
<td>Char</td>
<td>N/A</td>
<td>32</td>
<td>‘\0’</td>
<td>‘\Uffffffff’</td>
</tr>
</tbody>
</table>

Char natively supports representation of Unicode characters; see Strings for more details.

• Floating-point types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision</th>
<th>Number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float16</td>
<td>half</td>
<td>16</td>
</tr>
<tr>
<td>Float32</td>
<td>single</td>
<td>32</td>
</tr>
<tr>
<td>Float64</td>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>

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:

```julia
# 32-bit system:
julia> Int
Int32
julia> Uint
Uint32

# 64-bit system:
julia> Int
Int64
julia> Uint
Uint64
```

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

```julia
# 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
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
julia> 0b10
0x02
```
The minimum and maximum representable values of primitive numeric types such as integers are given by the `typemin` and `typemax` functions:

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

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
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.

### 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
1e+10

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
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:

<table>
<thead>
<tr>
<th>Special value</th>
<th>Float16</th>
<th>Float32</th>
<th>Float64</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inf16</td>
<td>Inf32</td>
<td>Inf</td>
<td>Inf</td>
<td>positive infinity</td>
<td>a value greater than all finite floating-point values</td>
</tr>
<tr>
<td>-Inf16</td>
<td>-Inf32</td>
<td>-Inf</td>
<td>-Inf</td>
<td>negative infinity</td>
<td>a value less than all finite floating-point values</td>
</tr>
<tr>
<td>NaN16</td>
<td>NaN32</td>
<td>NaN</td>
<td>NaN</td>
<td>not a number</td>
<td>a value not == to any floating-point value (including itself)</td>
</tr>
</tbody>
</table>

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))
(Float16(0xfc00),Float16(0x7c00))

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

julia> eps(Float32)
1.192092896e-07

julia> eps(Float64)
2.22044604925031308e-16

julia> eps() #Same as eps(Float64)
2.22044604925031308e-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)
2.22044604925031308e-16

julia> eps(1000.)
1.13686837721616030e-13

julia> eps(1e-27)
1.79366203433576585e-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, \( \text{eps}(1.0) \) is the same as \( \text{eps}(\text{Float64}) \) 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
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
julia> 1.1 + 0.1
1.2000000000000002

julia> with_rounding(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.*

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

julia> BigInt("123456789012345678901234567890") + 1
123456789012345678901234567891

julia> BigFloat("1.23456789012345678901")
1.234567890123456789010000000000000000000000000000000000000004e+00 with 256 bits of precision

julia> BigFloat(2.0^66) / 3
2.45956587649460682133333333333333333333333333333333333333333333344e+19 with 256 bits of precision

julia> factorial(BigInt(40))
81591528324789773434561126959611894272000000000

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
```

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

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

julia> x = x - 1
9223372036854775807

julia> typeof(x)
Int64

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

julia> y = y - 1
-9223372036854775809

julia> typeof(y)
BigInt
```

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

### 1.4. Integers and Floating-Point Numbers
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
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
julia> 2^2x
64
```

The precedence of numeric literal coefficients is the same as that of unary operators such as negation. So $2^3x$ is parsed as $2^{(3x)}$, and $2x^3$ is parsed as $2*(x^3)$.

Numeric literals also work as coefficients to parenthesized expressions:

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

*type error*: apply: expected Function, got Int64

```julia
julia> x(x+1)
```

*type error*: 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 literal:

• 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero(x)</td>
<td>Literal zero of type x or type of variable x</td>
</tr>
<tr>
<td>one(x)</td>
<td>Literal one of type x or type of variable x</td>
</tr>
</tbody>
</table>

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

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 func-
tions.
### 1.5.1 Arithmetic Operators

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

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

As well as the negation on `Bool` types:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!x</td>
<td>negation</td>
<td>changes <code>true</code> to <code>false</code> and vice versa</td>
</tr>
</tbody>
</table>

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
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>~x</td>
<td>bitwise not</td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x $ y</td>
<td>bitwise xor (exclusive or)</td>
</tr>
<tr>
<td>x &gt;&gt; y</td>
<td>logical shift right</td>
</tr>
<tr>
<td>x &gt;&gt; y</td>
<td>arithmetic shift right</td>
</tr>
<tr>
<td>x &lt;&lt; y</td>
<td>logical/arithmetic shift left</td>
</tr>
</tbody>
</table>

Here are some examples with bitwise operators:

```julia
julia> ~123
-124
julia> 123 & 234
106
julia> 123 | 234
251
```
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 `=` 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:

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

1.5.4 Numeric Comparisons

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equality</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>inequality</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

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.
- $\text{Inf}$ is equal to itself and greater than everything else except $\text{NaN}$.
- $-\text{Inf}$ is equal to itself and less than everything else except $\text{NaN}$.
- $\text{NaN}$ 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Tests if</th>
</tr>
</thead>
<tbody>
<tr>
<td>isequal(x, y)</td>
<td>x and y are identical</td>
</tr>
<tr>
<td>isfinite(x)</td>
<td>x is a finite number</td>
</tr>
<tr>
<td>isinf(x)</td>
<td>x is infinite</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>x is not a number</td>
</tr>
</tbody>
</table>

isequal considers NaNs equal to each other:
julia> isequal(NaN,NaN)
true

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

julia> isequal(NaN,NaN32)
false

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 very tricky. A great deal of care has been taken to ensure that Julia does them correctly.

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 element-wise comparisons, which allows them to work on arrays. For example, 0 .< A .< 1 gives a boolean array whose entries are true where the corresponding elements of A are between 0 and 1.

Note the evaluation behavior of chained comparisons:

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).

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round(x)</code></td>
<td>round x to the nearest integer</td>
<td>FloatingPoint</td>
</tr>
<tr>
<td><code>iround(x)</code></td>
<td>round x to the nearest integer</td>
<td>Integer</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>round x towards -Inf</td>
<td>FloatingPoint</td>
</tr>
<tr>
<td><code>ifloor(x)</code></td>
<td>round x towards -Inf</td>
<td>Integer</td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>round x towards +Inf</td>
<td>FloatingPoint</td>
</tr>
<tr>
<td><code>iceil(x)</code></td>
<td>round x towards +Inf</td>
<td>Integer</td>
</tr>
<tr>
<td><code>trunc(x)</code></td>
<td>round x towards zero</td>
<td>FloatingPoint</td>
</tr>
<tr>
<td><code>itrunc(x)</code></td>
<td>round x towards zero</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Division functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div(x,y)</code></td>
<td>truncated division; quotient rounded towards zero</td>
</tr>
<tr>
<td><code>fld(x,y)</code></td>
<td>floored division; quotient rounded towards −Inf</td>
</tr>
<tr>
<td><code>rem(x,y)</code></td>
<td>remainder; satisfies $x == \text{div}(x,y) \times y + \text{rem}(x,y)$; sign matches $x$</td>
</tr>
<tr>
<td><code>mod(x,y)</code></td>
<td>modulus; satisfies $x == \text{fld}(x,y) \times y + \text{mod}(x,y)$; sign matches $y$</td>
</tr>
<tr>
<td><code>gcd(x,y...)</code></td>
<td>greatest common divisor of $x, y,...$; sign matches $x$</td>
</tr>
<tr>
<td><code>lcm(x,y...)</code></td>
<td>least common multiple of $x, y,...$; sign matches $x$</td>
</tr>
</tbody>
</table>

Sign and absolute value functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>a positive value with the magnitude of $x$</td>
</tr>
<tr>
<td><code>abs2(x)</code></td>
<td>the squared magnitude of $x$</td>
</tr>
<tr>
<td><code>sign(x)</code></td>
<td>indicates the sign of $x$, returning -1, 0, or +1</td>
</tr>
<tr>
<td><code>signbit(x)</code></td>
<td>indicates whether the sign bit is on (1) or off (0)</td>
</tr>
<tr>
<td><code>copysign(x,y)</code></td>
<td>a value with the magnitude of $x$ and the sign of $y$</td>
</tr>
<tr>
<td><code>flipsign(x,y)</code></td>
<td>a value with the magnitude of $x$ and the sign of $x \times y$</td>
</tr>
</tbody>
</table>

Powers, logs and roots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqrt(x)</code></td>
<td>the square root of $x$</td>
</tr>
<tr>
<td><code>cbrt(x)</code></td>
<td>the cube root of $x$</td>
</tr>
<tr>
<td><code>hypot(x,y)</code></td>
<td>hypotenuse of right-angled triangle with other sides of length $x$ and $y$</td>
</tr>
<tr>
<td><code>exp(x)</code></td>
<td>the natural exponential function at $x$</td>
</tr>
<tr>
<td><code>expm1(x)</code></td>
<td>accurate $\exp(x) - 1$ for $x$ near zero</td>
</tr>
<tr>
<td><code>ldexp(x,n)</code></td>
<td>$x \times 2^n$ computed efficiently for integer values of $n$</td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>the natural logarithm of $x$</td>
</tr>
<tr>
<td><code>log(b,x)</code></td>
<td>the base $b$ logarithm of $x$</td>
</tr>
<tr>
<td><code>log2(x)</code></td>
<td>the base 2 logarithm of $x$</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>the base 10 logarithm of $x$</td>
</tr>
<tr>
<td><code>log1p(x)</code></td>
<td>accurate $\log(1+x)$ for $x$ near zero</td>
</tr>
<tr>
<td><code>exponent(x)</code></td>
<td>returns the binary exponent of $x$</td>
</tr>
<tr>
<td><code>significand(x)</code></td>
<td>returns the binary significand (a.k.a. mantissa) of a floating-point number $x$</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
\sin &\quad \cos &\quad \tan &\quad \cot &\quad \sec &\quad \csc \\
\sinh &\quad \cosh &\quad \tanh &\quad \coth &\quad \operatorname{sech} &\quad \operatorname{csch} \\
\arcsin &\quad \arccos &\quad \arctan &\quad \arccot &\quad \arccosec &\quad \arccsc \\
\operatorname{arsinh} &\quad \operatorname{arcosh} &\quad \operatorname{artanh} &\quad \operatorname{arcoth} &\quad \operatorname{arsech} &\quad \operatorname{arcsch} \\
sinc &\quad \cosc &\quad \arctan2
\end{align*}
\]

These are all single-argument functions, with the exception of \(\arctan2\), which gives the angle in radians between the \(x\)-axis and the point specified by its arguments, interpreted as \(x\) and \(y\) coordinates.

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

\[
\begin{align*}
\sin_d &\quad \cos_d &\quad \tan_d &\quad \cot_d &\quad \sec_d &\quad \csc_d \\
\arcsin_d &\quad \arccos_d &\quad \arctan_d &\quad \arccot_d &\quad \arccosec_d &\quad \arccsc_d
\end{align*}
\]

Special functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\operatorname{erf}(x))</td>
<td>the error function at (x)</td>
</tr>
<tr>
<td>(\operatorname{erfc}(x))</td>
<td>the complementary error function, i.e. the accurate version of (1-\operatorname{erf}(x)) for large (x)</td>
</tr>
<tr>
<td>(\operatorname{erfinv}(x))</td>
<td>the inverse function to (\operatorname{erf})</td>
</tr>
<tr>
<td>(\operatorname{erfcinv}(x))</td>
<td>the inverse function to (\operatorname{erfc})</td>
</tr>
<tr>
<td>(\operatorname{erfi}(x))</td>
<td>the imaginary error function defined as (-\text{i}m \times \operatorname{erf}(x \times \text{i}m)), where \text{i}m is the imaginary unit</td>
</tr>
<tr>
<td>(\operatorname{erfxc}(x))</td>
<td>the scaled complementary error function, i.e. accurate (\exp(x^2) \times \operatorname{erfc}(x)) for large (x)</td>
</tr>
<tr>
<td>(\operatorname{dawson}(x))</td>
<td>the scaled imaginary error function, a.k.a. Dawson function, i.e. accurate (\exp(-x^2) \times \operatorname{erfi}(x)) for large (x)</td>
</tr>
<tr>
<td>(\operatorname{gamma}(x))</td>
<td>the gamma function at (x)</td>
</tr>
<tr>
<td>(\operatorname{lgamma}(x))</td>
<td>accurate (\log(\operatorname{gamma}(x))) for large (x)</td>
</tr>
<tr>
<td>(\operatorname{lfcfact}(x))</td>
<td>accurate (\log(\text{factorial}(x))) for large (x); same as (\operatorname{lgamma}(x+1)) for (x &gt; 1), zero otherwise</td>
</tr>
<tr>
<td>(\operatorname{digamma}(x))</td>
<td>the digamma function (i.e. the derivative of (\operatorname{lgamma})) at (x)</td>
</tr>
<tr>
<td>(\operatorname{beta}(x,y))</td>
<td>the beta function at (x, y)</td>
</tr>
<tr>
<td>(\operatorname{lbeta}(x,y))</td>
<td>accurate (\log(\operatorname{beta}(x, y))) for large (x) or (y)</td>
</tr>
<tr>
<td>(\operatorname{eta}(x))</td>
<td>the Dirichlet eta function at (x)</td>
</tr>
<tr>
<td>(\operatorname{zeta}(x))</td>
<td>the Riemann zeta function at (x)</td>
</tr>
<tr>
<td>(\operatorname{airy}(x), \operatorname{airyai}(x))</td>
<td>the Airy Ai function at (x)</td>
</tr>
<tr>
<td>(\operatorname{airyprime}(x), \operatorname{airyaiprime}(x))</td>
<td>the derivative of the Airy Ai function at (x)</td>
</tr>
<tr>
<td>(\operatorname{airybi}(x))</td>
<td>the Airy Bi function at (x)</td>
</tr>
<tr>
<td>(\operatorname{airybiprime}(x))</td>
<td>the derivative of the Airy Bi function at (x)</td>
</tr>
<tr>
<td>(\operatorname{airy}(k,x))</td>
<td>the (k)-th derivative of the Airy Ai function at (x)</td>
</tr>
<tr>
<td>(\operatorname{besselj}(nu,z))</td>
<td>the Bessel function of the first kind of order (nu) at (z)</td>
</tr>
<tr>
<td>(\operatorname{besselj0}(z))</td>
<td>(\operatorname{besselj}(0,z))</td>
</tr>
<tr>
<td>(\operatorname{besselj1}(z))</td>
<td>(\operatorname{besselj}(1,z))</td>
</tr>
<tr>
<td>(\operatorname{bessely}(nu,z))</td>
<td>the Bessel function of the second kind of order (nu) at (z)</td>
</tr>
<tr>
<td>(\operatorname{bessely0}(z))</td>
<td>(\operatorname{bessely}(0,z))</td>
</tr>
<tr>
<td>(\operatorname{bessely1}(z))</td>
<td>(\operatorname{bessely}(1,z))</td>
</tr>
<tr>
<td>(\operatorname{besselh}(nu,k,z))</td>
<td>the Bessel function of the third kind (a.k.a. Hankel function) of order (nu) at (z); (k) must be either (1) or (2)</td>
</tr>
<tr>
<td>(\operatorname{hankelh1}(nu,z))</td>
<td>(\operatorname{besselh}(nu, 1, z))</td>
</tr>
<tr>
<td>(\operatorname{hankelh2}(nu,z))</td>
<td>(\operatorname{besselh}(nu, 2, z))</td>
</tr>
<tr>
<td>(\operatorname{besseli}(nu,z))</td>
<td>the modified Bessel function of the first kind of order (nu) at (z)</td>
</tr>
<tr>
<td>(\operatorname{besselk}(nu,z))</td>
<td>the modified Bessel function of the second kind of order (nu) at (z)</td>
</tr>
</tbody>
</table>
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
julia> 1 + 2im
1 + 2im
```

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

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

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

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

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

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

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

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

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

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

julia> 3(2 - 5im)^-1.0
0.20689655172413793 + 0.5172413793103449im
```

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

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

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

julia> (1 + 2im) + 0.5
```

22 Chapter 1. The Julia Manual
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

As is common, the absolute value of a complex number is its distance from zero. The abs2 function gives the square of the absolute value, and is of particular use for complex numbers, where it avoids taking a square root. The full gamut of other Elementary Functions is also defined for complex numbers:

julia> sqrt(im)
0.7071067811865476 + 0.7071067811865475im

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

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991517997im

julia> exp(1 + 2im)
-1.1312043837568138 + 2.471726672004819im

julia> sinh(1 + 2im)
-0.48905625904129374 + 1.4031192506220407im

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

```
julia> sqrt(-1)
ERROR: DomainError()
    in sqrt at math.jl:111
```

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

The *literal numeric coefficient notation* does 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
complex(1.0,Inf)
```

```
julia> 1 + NaN*im
complex(1.0,Nan)
```

### 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 Inf

julia> -3//0 -Inf

julia> typeof(ans) Rational{Int64}

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

julia> 0//0 invalid rational: 0//0
As usual, the promotion system makes interactions with other numeric types effortless:

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

julia> 3//5 - 0.5
0.1

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

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

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

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

julia> 1 + 2//3im
1//1 + 2//3im

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.003333333333332993
```

## 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
julia> 'x'
'x'

julia> typeof(ans)
Char
```

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

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

julia> typeof(ans)
Int64
```

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

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

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
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 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('7')
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

Here a variable is initialized with a simple string literal:
If you want to extract a character from a string, you index into it:

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

julia> str[end/2]
','

julia> str[end/3]
'o'

julia> str[end/4]
'l'
```

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

```julia
julia> str[0]
BoundsError()

julia> str[end+1]
BoundsError()
```

You can also extract a substring using range indexing:

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

Note the distinction between `str[k]` and `str[k:k]`:

```julia
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]
invalid UTF-8 character index

julia> s[3]
invalid UTF-8 character index

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 character 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 do 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, but 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 the generic string function to convert objects into String form. Most non-String objects are converted to strings as they are shown in interactive sessions:

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

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

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

julia> c = ‘x’
‘x’
To include a literal $ in a string literal, escape it with a backslash:

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

### 1.7.5 Common Operations

You can lexicographically compare strings using the standard comparison operators:

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

julia> "abracadabra" == "xylophone"
false

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

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

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

```julia
julia> search("xylophone", 'x')
1

julia> search("xylophone", 'p')
5

julia> search("xylophone", 'z')
0
```

You can start the search for a character at a given offset by providing a third argument:

```julia
julia> search("xylophone", 'o')
4

julia> search("xylophone", 'o', 5)
7

julia> search("xylophone", 'o', 8)
0
```

Another handy string function is `repeat`:

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

Some other useful functions include:

- `endof(str)` gives the maximal (byte) index that can be used to index into `str`.
- `length(str)` the number of characters in `str`.
- `i = start(str)` gives the first valid index at which a character can be found in `str` (typically 1).
• `c, j = next(str, i)` returns next character at or after the index `i` and the next valid character index following that. With `start` and `eof`, can be used to iterate through the characters in `str`.
• `ind2chr(str, i)` gives the number of characters in `str` up to and including any at index `i`.
• `chr2ind(str, j)` gives the index at which the `j`th character in `str` occurs.

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, as described below, are one example of a non-standard string literal. 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*(?:#|\$)"
```

To check if a regex matches a string, use the `ismatch` function:

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

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 a pair of somewhat contrived examples:

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

julia> m.match
"acd"

julia> m.captures
3-element Union(UTF8String,ASCIIString,Nothing) Array:
  "a"
  "c"
  "d"

julia> m.offset
1

julia> m.offsets
3-element Int64 Array:
 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 Union(UTF8String,ASCIIString,Nothing) Array:
  "a"
  nothing
  "d"

julia> m.offset
1

julia> m.offsets
3-element Int64 Array:
 1
```
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
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 `perldre` 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
julia> r"a+.b+.?d$s"ism
r"a+.b+.?d$s"ims

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

### 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.

---

**1.7. Strings**
• \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
julia> b"DATA\xff\u2200"
[68,65,84,65,255,226,136,128]
```

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
julia> "DATA\xff\u2200"
syntax error: 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
julia> b"\xff"
1-element Uint8 Array:
0xff
julia> b"\uff"
2-element Uint8 Array:
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.

## 1.8 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:

```julia
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
```

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

### 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 functions definitions can be entered into interactive sessions, it is easy to compare these definitions:

```markdown
\[ f(x, y) = x + y \]
```

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

```
 julia> f(2, 3)
  5
```
Of course, in a purely linear function body like `g`, the usage of `return` is pointless since the expression `x + y` is never evaluated and we could simply make `x * y` the last expression in the function and omit the `return`. In conjunction with other control flow, however, `return` is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length `x` and `y`, avoiding overflow:

```julia
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 with any other function:

```julia
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
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:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A B C ...]</td>
<td>hcat</td>
</tr>
<tr>
<td>[A, B, C, ...]</td>
<td>vcat</td>
</tr>
<tr>
<td>[A B; C D; ...</td>
<td>hvcat</td>
</tr>
<tr>
<td>A'</td>
<td>ctranspose</td>
</tr>
<tr>
<td>A.'</td>
<td>transpose</td>
</tr>
<tr>
<td>1:n</td>
<td>colon</td>
</tr>
<tr>
<td>A[i]</td>
<td>getindex</td>
</tr>
<tr>
<td>A[i]=x</td>
<td>setindex!</td>
</tr>
</tbody>
</table>

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
 #<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 the `map` function, 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 Float64 Array:
   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 Int64 Array:
   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 destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

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

### 1.8. Functions
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:

```
function bar(a,b,x...) = (a,b,x)
```

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

```julia
julia> x = [3,4]
2-element Int64 Array:
3
4
julia> bar(1,2,x...)  
(1,2,(3,4))

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

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

julia> args = [1,2]
2-element Int64 Array:
1
2
julia> baz(args...)  
3

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

```julia
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
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:

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

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

```julia
function f(x; 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, `f(x; k...)`. 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. For example, given this definition:

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

the b in a=b refers to the 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 a=b 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 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:

```julia
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:

```julia
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`. 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, the standard library provides a function `cd` for running code in a given directory, and switching back to the previous directory when the code finishes or aborts. There is also a definition of `open` that runs code ensuring that the opened file is eventually closed. We can combine these functions to safely write a file in a certain directory:

```julia
cd("data") do
    open("outfile", "w") do f
        write(f, data)
    end
end
```

The function argument to `cd` takes no arguments; it is just a block of code. The function argument to `open` receives a handle to the opened file.
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;
```
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:

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

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression is true, the else block is evaluated. Here it is in action:

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

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.

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

```julia
julia> if 1
    println("true")
end

 type error: lambda: in if, expected Bool, got Int64
```

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:
The expression `a ? b : c` 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
julia> x = 1; y = 2;
println(x < y ? "less than" : "not less than")  # less than

julia> x = 1; y = 0;
println(x < y ? "less than" : "not less than")  # not less than
```

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

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

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

```julia
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.

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:

1.9. Control Flow
Julia Language Documentation, Release 0.2.0

julia> f(1) & t(2)
1
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 is an error:

julia> 1 && 2

error: lambda: in if, expected Bool, got Int64

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 < n 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
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
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
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
```

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

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

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

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

### 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
ArgumentError
BoundsError
DivideError
DomainError
EOFError
ErrorException
InexactError
InterruptException
KeyError
LoadError
MemoryError
MethodError
OverflowError
ParseError
SystemError
TypeError
UndefRefError

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

```
julia> sqrt(-1)
ERROR: DomainError()
in sqrt at math.jl:117
```

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

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)
negative x not allowed

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
negative x not allowed

**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
MESSAGE: 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

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
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
julia> sqrt_second([1 4])
2.0
```

```julia
julia> sqrt_second([1 -4])
0.0 + 2.0im
```

```julia
julia> sqrt_second(9)
3.0
```

```julia
julia> sqrt_second(-9)
ERROR: DomainError()
    in sqrt at math.jl:117
    in sqrt_second at none:7
```

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:

```julia
f = open("file")
try
    # operate on file f
finally
```
```julia
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
julia> p = Task(producer)
Task

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
6

julia> consume(p)
```

---

54 Chapter 1. The Julia Manual
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 are intended for multitasking, and do not launch threads that can run on separate CPUs. True kernel threads are discussed under the topic of Parallel Computing.

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, 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. These constructs which 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 explicitly 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:

```julia
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:

```julia
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 \( x \)):

```julia
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:

```julia
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:

```julia
even(n) = n == 0 ? true : odd(n-1)
odd(n) = n == 0 ? false : even(n-1)
```

```julia
julia> even(3)
false
julia> odd(3)
true
```

Julia provides built-in, efficient functions to test this 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
julia> for i = 1:1; y = 10; end

julia> y
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.

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:

```julia
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:

```julia
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:

```julia
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
julia> begin
    local x = 1
    begin
      local x = 2
      end
    end
syntax error: local x declared twice

julia> begin
    local x = 1
    let
      local x = 2
    end
```
The first example is illegal because you cannot declare the same variable as local in the same scope twice. The second example is legal 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:

```julia
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:

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

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

```julia
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:

```julia
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, parametric and dependent. 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 paramaterized by other types and by certain other values (currently integers and symbols). Type parameters may be completely 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

The :: operator is read as “is an instance of” and 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.

When attached to a variable, 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 the convert function:

```
julia> function foo()
   x::Int8 = 1000
   x
end

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
```

In value contexts, such as `f(x::Int8)`, the :: is a type assertion again and not a declaration. Note that these declarations cannot be used in global scope currently, in the REPL, since Julia does not yet have constant-type globals.

### 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:
The `abstract` keyword introduces a new abstract type, whose name is given by `«name»`. This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this “parent” type.

When no supertype is given, the default supertype is `Any` — a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, `Any` is commonly called “top” because it is at the apex of the type graph. Julia also has a predefined abstract “bottom” type, at the nadir of the type graph, which is called `None`. It is the exact opposite of `Any`: no object is an instance of `None` and all types are supertypes of `None`.

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

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

```julia
julia> Integer <: Number
true

julia> Integer <: FloatingPoint
false
```

Since abstract types have no instantiations and serve as no more than nodes in the type graph, there is not much more to say about them until we introduce parametric abstract types later on in Parametric Types.

### 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:

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

```julia
julia> typeof(foo)
Foo
```

Since the `bar` field is unconstrained in type, any value will do; the value for `baz` must be an `Int` and `qux` must be a `Float64`. The signature of the default constructor is taken directly from the field type declarations `(Any, Int, Float64)`, so arguments must match this implied type signature:

```julia
julia> Foo((), 23.5, 1)
no method Foo((), Float64, Int64)
```

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

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

```julia
julia> foo.baz
23
```

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

You can also change the values as one would expect:

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

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

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

```julia
type NoFields
end
```

```julia
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`:

```julia
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.

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

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

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

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

### 1.11. Types

---

65
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(Int,String)

 julia> 1 :: IntOrString
 1

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

 julia> 1.0 :: IntOrString
 type error: typeassert: expected Union(Int,String), 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, `None`:

```
 julia> Union()
 None
```

Recall from the discussion above that `None` 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. `None`.

### 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:

```julia
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
julia> Point{Float64}
Point{Float64}

julia> Point{String}
Point{String}
```

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
julia> Point
Point{T}
```

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
julia> Point{Float64} <: Point
true

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

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

```julia
julia> Float64 <: Point
false

julia> String <: Point
false
```
Concrete `Point` types with different values of `T` are never subtypes of each other:

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

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

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

julia> typeof(ans)
Point{Float64}
```

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

```julia
julia> Point{Float64}(1.0)
no method Point{Float64,}

julia> Point{Float64}(1.0,2.0,3.0)
no method Point{Float64,Float64,Float64}
```

The provided arguments need to match the field types exactly, in this case `(Float64,Float64)`, as with all composite type default constructors.

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

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

julia> Point(1,2)
Point(1,2)

julia> typeof(ans)
Point{Int64}

In the case of `Point`, the type of `T` is unambiguously implied if and only if the two arguments to `Point` have the same type. When this isn’t the case, the constructor will fail with a no method error:

julia> Point(1,2.5)
no method Point(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:

```julia
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:

```julia
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
```

1.11. Types
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 \( x = y \):

```julia
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:

```julia
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
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

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

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

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

```julia
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:

```julia
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” \( \text{Type}(T) \) is an abstract type whose only instance is the object \( T \). Since the definition is a little difficult to parse, let’s look at some examples:

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

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

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

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

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

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

julia> isa(Float64, Type)
true

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

Any object that is not a type is not an instance of \( \text{Type} \):

```julia
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:

```julia
# 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
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:

```julia
# 32-bit system:
julia> UInt
Uint32

# 64-bit system:
julia> UInt
Uint64
```

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

```julia
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 pre-defined 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
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:

```julia
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.


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
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
julia> typeof(Rational)
DataType

julia> typeof(Union(Real, Float64, Rational))
UnionType

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
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
julia> typeof(()
()

julia> typeof(DataType)
DataType

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

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

All fixed points of the typeof function 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 “no method” error is raised:

julia> super(Union(Float64, Int64))
nomethod super(UnionType,)

julia> super(None)
nomethod super(UnionType,)

julia> super((Float64, Int64))
nomethod super((DataType,DataType),)

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 very 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 very powerful and convenient paradigm for structuring and organizing programs.

¹ In C++ or Java, for example, in a method call like \texttt{obj.meth(arg1, arg2)}, the object \texttt{obj} “receives” the method call and is implicitly passed to the method via the \texttt{this} keyword, rather than as an explicit method argument. When the current \texttt{this} object is the receiver of a method call, it can be omitted altogether, writing just \texttt{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:

\[ f(x::\text{Float64}, y::\text{Float64}) = 2x + y \]

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)
no method f(Float64,Int64)

julia> f(float32(2.0), 3.0)
no method f(Float32,Float64)

julia> f(2.0, "3.0")
no method f(Float64,ASCIIString)

julia> f("2.0", "3.0")
no method 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:

\[ f(x::\text{Number}, y::\text{Number}) = 2x - y \]

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 \(f(\text{Float64},\text{Float64})\) method cannot be called and the more general \(f(\text{Number},\text{Number})\) method must be used:

julia> f(2.0, 3.0)
7.0
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)
 no method f(ASCIIString,Int64)
```

```
 julia> f()
 no method f()
```

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

```
 julia> f
 Methods for generic function f
 f(Float64,Float64)
 f(Number,Number)
```

This output tells us that \(f\) is a function object with two methods: one taking two \(\text{Float64}\) arguments and one taking arguments of type \(\text{Number}\).

In the absence of a type declaration with \(::\), the type of a method parameter is \(\text{Any}\) by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type \(\text{Any}\). 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(+)
 # 92 methods for generic function "+":
 +(x::Bool,y::Bool) at bool.jl:38
 +(x::Union(Array{Bool,N},SubArray{Bool,N,A<:Array{T,N},I<:(Union(Rangel{Int64},Int64,Range{Int64})))...,
 +(S,T)(A::Union(Array{S,N},SubArray{S,N,A<:Array{T,N},I<:(Union(Rangel{Int64},Int64,Range{Int64}))...,
 +(T::Union((Int16,Int8,Int32)),x::T::Union((Int16,Int8,Int32)),y::T::Union((Int16,Int8,Int32))) at int.jl:10
 +(T::Union((Uint16,Uint8,Uint32)),x::T::Union((Uint16,Uint8,Uint32)),y::T::Union((Uint16,Uint8,Uint32))) at int.jl:11
 +(x::Int64,y::Int64) at int.jl:41
 +(x::Uint64,y::Uint64) at int.jl:42
 +(x::Int128,y::Int128) at int.jl:43
 +(x::Uint128,y::Uint128) at int.jl:44
 +(a::Float16,b::Float16) at float.jl:129
```
Methods

1.12. Methods
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
julia> g(x::Float64, y) = 2x + y
julia> g(x, y::Float64) = x + 2y
```

Warning: New definition `g(Any,Float64)` is ambiguous with `g(Float64,Any)`.

To fix, define `g(Float64,Float64)` before the new definition.

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

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
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
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]

Julia> myappend([1,2,3],4)
4-element Int64 Array:
1
2
3
4
julia> myappend([1,2,3],2.5)
no method myappend(Array{Int64,1},Float64)

julia> myappend([1.0,2.0,3.0],4.0)
[1.0,2.0,3.0,4.0]

julia> myappend([1.0,2.0,3.0],4)
no method 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

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:
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

Constructors\(^1\) are functions that create new objects — specifically, instances of *Composite Types*. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```julia
struct Foo
    bar
    baz
end

julia> foo = Foo(1,2)  # Create a new Foo object
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:

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

```julia
julia> Foo(1)
Foo(1,1)
```

You could also add a zero-argument `Foo` constructor method that supplies default values for both the `bar` and `baz` fields:

```julia
1.13. Constructors 81

\(^1\) Nomenclature: while the term “constructor” generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as “constructors”. In such situations, it is generally clear from context that the term is used to mean “constructor method” rather than “constructor function”, especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.
Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called outer constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default one.

### 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:

```julia
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 <= y`:

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

julia> OrderedPair(2,1)
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:
This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent — one with a default constructor, the other with an explicit constructor:

```julia
type T1
    x::Int64
end

type T2
    x::Int64
    T2(x::Int64) = new(x)
end
```

```julia
julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
no method T1(Float64,

julia> T2(1.0)
no method T2(Float64,
```

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:

```julia
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:

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

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.
To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, we take another crack at defining the `SelfReferential` type, with a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

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

```julia
julia> is(x, x.obj)
true
```

```julia
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
julia> z = Incomplete();
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized field is an immediate error:

```julia
julia> z.xx
access to undefined reference
```

This prevents uninitialized fields from propagating throughout a program or forcing programmers to continually check for uninitialized fields, the way they have to check for `null` values everywhere in Java: if a field is uninitialized and it is used in any way, an error is thrown immediately so no error checking is required. You can also pass incomplete objects to other functions from inner constructors to delegate their completion:

```julia
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(1,2)

julia> Point(1.0,2.5)
Point(1.0,2.5)

julia> Point(1,2.5)
no method Point(Int64, Float64)

## explicit T ##
julia> Point{Int64}(1,2)
Point(1,2)

julia> Point{Int64}(1.0,2.5)
no method Point(Float64, Float64)

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

julia> Point{Float64}(1,2)
no method Point(Int64, Int64)
```

As you can see, for constructor calls with explicit type parameters, the arguments must match that specific type: `Point{Int64}(1,2)` works, but `Point{Int64}(1.0,2.5)` does not. When the type is implied by the arguments to the constructor call, as in `Point(1,2)`, then the types of the arguments must agree — otherwise the `T` cannot be determined — but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What’s really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type `T`. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`, that behaves just like non-parametric default inner constructors do. It also provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

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

    Point(x::T, y::T) = 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 declara-
tions always define methods of \texttt{Point(T)} rather than methods of the general \texttt{Point} constructor function. Since \texttt{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 \texttt{Point(x::T, y::T) = new(x, y)} provides an inner constructor method for each value of \texttt{T}. It is thus this method declaration that defines the behavior of constructor calls with explicit type parameters like \texttt{Point\{Int64\}(1,2)} and \texttt{Point\{Float64\}(1.0,2.0)}. The outer constructor declaration, on the other hand, defines a method for the general \texttt{Point} constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like \texttt{Point(1,2)} and \texttt{Point(1.0,2.5)}, work. Since the method declaration restricts the arguments to being of the same type, calls like \texttt{Point(1,2.5)}, with arguments of different types, result in “no method” errors.

Suppose we wanted to make the constructor call \texttt{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:

\begin{verbatim}
Point(x::Int64, y::Float64) = Point(convert(Float64,x),y)
\end{verbatim}

This method uses the \texttt{convert} function to explicitly convert \texttt{x} to \texttt{Float64} and then delegates construction to the general constructor for the case where both arguments are \texttt{Float64}. With this method definition what was previously a “no method” error now successfully creates a point of type \texttt{Point\{Float64\}}:

\begin{verbatim}
julia> Point(1,2.5)
Point(1.0,2.5)
\end{verbatim}

However, other similar calls still don’t work:

\begin{verbatim}
julia> Point(1.5,2)
no method Point\{Float64,Int64\}
\end{verbatim}

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 the all it takes is the following outer method definition to make all calls to the general \texttt{Point} constructor work as one would expect:

\begin{verbatim}
Point(x::Real, y::Real) = Point(promote(x,y)...)
\end{verbatim}

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

\begin{verbatim}
julia> Point(1.5,2)
Point(1.5,2.0)

julia> Point(1,1//2)
Point(1//1,1//2)

julia> Point(1.0,1//2)
Point(1.0,0.5)
\end{verbatim}

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 \texttt{rational.jl}, which implements Julia’s \texttt{Rational Numbers}:
The first line — type 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:
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, 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)
12
julia> typeof(ans)
UInt8
julia> convert(FloatingPoint, 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(FloatingPoint, "foo")
no method convert(Type{FloatingPoint},ASCIIString)
```

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`. 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)
true
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:

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

```julia
julia> convert(Bool, 1im)
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:

```julia
convert{T<:Int} (::Type{Rational{T}}, x::Rational) = Rational(convert(T,x.num),convert(T,x.den))
convert{T<:Int} (::Type{Rational{T}}, x::Int) = Rational(convert(T,x), convert(T,1))
function convert{T<:Int} (::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
    convert{T<:Int} (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<:Int} (::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//1im,3//4 + 0//1im)
```

Integer values are promoted to the largest type of the integer values. Floating-point values are promoted to largest of the floating-point types. 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(15),int32(-5))
-3//1

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

```julia
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:

```julia
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
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:

```julia
promote_rule{T<:Int}(::Type{Rational{T}}, ::Type{T}) = Rational{T}
promote_rule{T<:Int,S<:Int}(::Type{Rational{T}}, ::Type{S}) = Rational{promote_type(T,S)}
promote_rule{T<:Int,S<:Int}(::Type{Rational{T}}, ::Type{Rational{S}}) = Rational{promote_type(T,S)}
promote_rule{T<:Int,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

    export MyType, foo

    type MyType
        x
    end

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

    import Base.show
    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 in `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`.

Once a variable is imported this way (or, equivalently, with the `import` keyword), 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.

Method definitions are a bit special: they do not search modules named in `using` statements. The definition `function foo()` creates a new `foo` in the current module, unless `foo` has already been imported from elsewhere. For example, in `MyModule` above we wanted to add a method to the standard `show` function, so we had to write `import Base.show`.

1.15.1 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

1.15.2 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.

1.15.3 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
1.15.4 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:

```julia
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.

1.15.5 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.

Macros must be exported if they are intended to be used outside their defining module. Macro names are written with `@` in import and export statements, e.g. `import 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 is homoiconic: it 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, as compared to preprocessor “macro” systems, like that of C and C++, that perform superficial textual manipulation as a separate pass before any real parsing or interpretation occurs. Another aspect of metaprogramming is reflection: the ability of a running program to dynamically discover properties of itself. Reflection emerges naturally from the fact that all data types and code are represented by normal Julia data structures, so the structure of the program and its types can be explored programmatically just like any other data.
### 1.16.1 Expressions and Eval

Julia code is represented as a syntax tree built out of Julia data structures of type `Expr`. This makes it easy to construct and manipulate Julia code from within Julia, without generating or parsing source text. Here is the definition of the `Expr` type:

```plaintext
type Expr
    head::Symbol
    args::Array{Any,1}
    typ
end
```

The `head` is a symbol identifying the kind of expression, and `args` is an array of subexpressions, which may be symbols referencing the values of variables at evaluation time, may be nested `Expr` objects, or may be actual values of objects. The `typ` field is used by type inference to store type annotations, and can generally be ignored.

There is special syntax for “quoting” code (analogous to quoting strings) that makes it easy to create expression objects without explicitly constructing `Expr` objects. There are two forms: a short form for inline expressions using `:` followed by a single expression, and a long form for blocks of code, enclosed in `quote ... end`. Here is an example of the short form used to quote an arithmetic expression:

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

julia> typeof(ex)
Expr

julia> ex.head
call

julia> typeof(ans)
Symbol

julia> ex.args
4-element Any Array:
    +
    a
    :(*{b,c})
    1

julia> typeof(ex.args[1])
Symbol

julia> typeof(ex.args[2])
Symbol

julia> typeof(ex.args[3])
Expr

julia> typeof(ex.args[4])
Int64
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can easily 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.

Here’s an example of the longer expression quoting form:

```julia
julia> quote
    x = 1
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can easily 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.
Symbols

When the argument to : is just a symbol, a Symbol object results instead of an Expr:

```julia
julia> :foo
foo
julia> typeof(ans)
Symbol
```

In the context of an expression, symbols are used to indicate access to variables, and when an expression is evaluated, a symbol evaluates to the value bound to that symbol in the appropriate scope.

Sometimes extra parentheses around the argument to : are needed to avoid ambiguity in parsing:

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

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

Symbols can also be created using the symbol function, which takes a character or string as its argument:

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

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

eval and Interpolation

Given an expression object, one can cause Julia to evaluate (execute) it at the top level scope — i.e. in effect like loading from a file or typing at the interactive prompt — using the eval function:

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

julia> eval(ans)
3

julia> ex = :(a + b)
+(a,b)

julia> eval(ex)
a not defined

julia> a = 1; b = 2;
```
Expressions passed to `eval` are not limited to returning values — they can also have side-effects that alter the state of the top-level evaluation environment:

```
 julia> ex = :(x = 1)
x = 1
 julia> x
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 top-level variable `x`.

Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, one can, from within Julia code, 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`.

Constructing `Expr` objects like this is powerful, but somewhat tedious and ugly. Since the Julia parser is already excellent at producing expression objects, Julia allows “splicing” or interpolation of expression objects, prefixed with `$`, into quoted expressions, written using normal syntax. The above example can be written more clearly and concisely using interpolation:

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

This syntax is automatically rewritten to the form above where we explicitly called `Expr`. 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.
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
 unsupported or misplaced expression $
```

1.16.2 Macros

Macros are the analogue of functions for expression generation at compile time: they allow the programmer to automatically generate expressions by transforming zero or more argument expressions into a single result expression, which then takes the place of the macro call in the final syntax tree. 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, ...)  
```
Before the program runs, this statement will be replaced with the result of calling an expander function for name on the expression arguments. Expanders are defined with the macro keyword:

```julia
macro name(expr1, expr2, ...)
  ...
end
```

Here, for example, is the definition of Julia’s @assert macro (see error.jl):

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

This macro can be used like this:

```julia
julia> @assert 1==1.0
julia> @assert 1==0
Assertion failed: 1==0
```

Macro calls are expanded so that the above calls are precisely equivalent to writing:

```julia
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. Therefore, if the test expression is true when evaluated, the entire expression evaluates to nothing, whereas if the test expression 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 and not the expression that computed it would be available.

The @assert example also shows how macros can include a quote block, which allows for convenient manipulation of expressions inside the macro body.

### Hygiene

An issue that arises in more complex macros is that of hygiene. In short, Julia must ensure that variables introduced and used by macros do not accidentally clash with the variables used in code interpolated into those macros. 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.

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:

```julia
macro time(ex)
  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 \texttt{ex} also contained assignments to a variable called \texttt{t0}, or defined its own \texttt{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 \texttt{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 \texttt{time} and \texttt{println} will refer to the standard library definitions.

One problem remains however. Consider the following use of this macro:

```julia
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

Here the user expression \texttt{ex} is a call to \texttt{time}, but not the same \texttt{time} function that the macro uses. It clearly refers to \texttt{MyModule.time}. Therefore we must arrange for the code in \texttt{ex} to be resolved in the macro call environment. This is done by “escaping” the expression with the \texttt{esc} function:

```julia
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 \texttt{x} to zero in the call environment:

```julia
macro zerox()
    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.

### Non-Standard String Literals

Recall from \textit{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 macros is just the following:
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:

```julia
for line = lines
    m = match(r"^\s*(?:#|$)"", line)
    if m.match == 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:

```julia
re = Regex("^\s*(?:#|$)"")
for line = lines
    m = match(re, line)
    if m.match == 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, has to 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. The vast majority of use cases, however, one does not construct regular expressions dynamically, depending on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is, well, 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:

```julia
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.
1.16.3 Reflection

In addition to the syntax-level introspection utilized in metaprogramming, Julia provides several other runtime reflection capabilities.

**Type fields** The names of data type fields (or module members) may be interrogated using the `names` command. For example, given the following type:

```julia
type Point
    x::FloatingPoint
    y
end
```

`names(Point)` will return the array: `Any[ :x :y ]`. Note that the type of each field in a `Point` is stored in the `types` field of the `Point` object:

```julia
julia> typeof(Point)
DataType
julia> Point.types
(FloatingPoint, Any)
```

**Subtypes** The direct subtypes of any `DataType` may be listed using `subtypes(t::DataType)`. For example, the abstract `DataType` `FloatingPoint` has four (concrete) subtypes:

```julia
julia> subtypes(FloatingPoint)
5-element Array{Any,1}:
    BigFloat
    Float16
    Float32
    Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive applications of `subtypes` allow to build the full type tree.

**Type internals** The internal representation of types is critically important when interfacing with C code. `isbits(T::DataType)` returns `true` if `T` is stored with C-compatible alignment. The offsets of each field may be listed using `fieldoffsets(T::DataType)`.

**Function methods** The methods of any function may be listed using `methods(f::Function)`.

**Function representations** Functions may be introspected at several levels of representation. The lowered form of a function is available using `code_lowered(f::Function, (Args...))`, and the type-inferred lowered form is available using `code_typed(f::Function, (Args...))`.

Closer to the machine, the LLVM Intermediate Representation of a function is printed by `code_llvm(f::Function, (Args...))`, and finally the resulting assembly instructions (after JIT’ing step) are available using `code_native(f::Function, (Args...))`.

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`.

1.17. Multi-dimensional Arrays
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 behaviour, should take care to create a copy of inputs that it may modify.

### 1.17.1 Arrays

#### Basic Functions

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

#### 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.
### Comprehensions

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

\[
A = \{ F(x,y,...) \text{ for } x=rx, y=ry, ... \}
\]

The meaning of this form is that \( F(x,y,...) \) is evaluated with the variables \( x, y, \text{etc.} \) 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 \( rx, ry, \text{etc.} \) 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
julia> const x = rand(8)
8-element Float64 Array:
0.276455
0.614847
0.0601373
0.896024
0.646236
0.143959
0.0462343
0.730987
```

---

**Function** | **Description**
--- | ---
\( \text{Array}(\text{type, dims...}) \) | an uninitialized dense array
\( \text{cell}(\text{dims...}) \) | an uninitialized cell array (heterogeneous array)
\( \text{zeros}(\text{type, dims...}) \) | an array of all zeros of specified type
\( \text{ones}(\text{type, dims...}) \) | an array of all ones of specified type
\( \text{true}(\text{dims...}) \) | a \text{Bool} array with all values \text{true}
\( \text{false}(\text{dims...}) \) | a \text{Bool} array with all values \text{false}
\( \text{reshape}(A, \text{dims...}) \) | an array with the same data as the given array, but with different dimensions.
\( \text{copy}(A) \) | copy \( A \)
\( \text{deepcopy}(A) \) | copy \( A \), recursively copying its elements
\( \text{similar}(A, \text{element_type, dims...}) \) | 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 \( A \) if omitted.
\( \text{reinterpret}(\text{type, A}) \) | an array with the same binary data as the given array, but with the specified element type
\( \text{rand}(\text{dims}) \) | Array of \text{Float64}s with random, iid and uniformly distributed values in \([0,1)\)
\( \text{randn}(\text{dims}) \) | Array of \text{Float64}s with random, iid and standard normally distributed random values
\( \text{eye}(n) \) | \( n \)-by-\( n \) identity matrix
\( \text{eye}(m, n) \) | \( m \)-by-\( n \) identity matrix
\( \text{linspace}(\text{start, stop, n}) \) | vector of \( n \) linearly-spaced elements from \text{start} to \text{stop}
\( \text{fill!}(A, x) \) | fill the array \( A \) with value \( x \)
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:

```julia
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
julia> { i/2 for i = 1:3 }
3-element Any Array:
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 (length(I_1), length(I_2), ..., length(I_n)), with location (i_1, i_2, ..., i_n) of X containing the value A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]. Trailing dimensions indexed with scalars are dropped. For example, the dimensions of A[I, 1] will be (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
julia> x = reshape(1:16, 4, 4)
4x4 Int64 Array
1 5 9 13
2 6 10 14
3 7 11 15
```
Assignment

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

\[ A[I_1, I_2, \ldots, I_n] = X \]

where each \( I_k \) may be:

1. A scalar value
2. A range of the form \( \ldots : a:b \) or \( \ldots : a:b:c \)
3. An arbitrary integer vector, including the empty vector \( \ldots [] \)
4. A boolean vector

If \( X \) is an array, its size must be \( (\text{length}(I_1), \text{length}(I_2), \ldots, \text{length}(I_n)) \), and the value in location \( i_1, i_2, \ldots, i_n \) of \( A \) is overwritten with the value \( X[I_1[i_1], I_2[i_2], \ldots, I_n[i_n]] \). 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!`:

\[ \text{setindex!}(A, X, I_1, I_2, \ldots, I_n) \]

Example:

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

 julia> x[1:2, 2:3] = -1
  3x3 Int64 Array
  1 -1 -1
  2 -1 -1
  3 6 9
```

Concatenation

Arrays can be concatenated along any dimension using the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat(k, A...)</td>
<td>concatenate input n-d arrays along the dimension k</td>
</tr>
<tr>
<td>vcat(A...)</td>
<td>shorthand for cat(1, A...)</td>
</tr>
<tr>
<td>hcat(A...)</td>
<td>shorthand for cat(2, A...)</td>
</tr>
<tr>
<td>hvcat(A...)</td>
<td></td>
</tr>
</tbody>
</table>

Concatenation operators may also be used for concatenating arrays:
### Vectorized Operators and Functions

The following operators are supported for arrays. In case of binary operators, the dot (element-wise) version of the operator should be used when both inputs are non-scalar, and any version of the operator may be used if one of the inputs is a scalar.

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

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

abs abs2 angle cbrt airy airyai airyaiprime airybi airybi prime airyprime acos acosh asin asinh atan atan2 atanh acsc acsch asec asech acot acoth cos cosh sin 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 besseljl besselj1 besselk bessely bessely0 bessely1 exp erf erfc erfcinv exp2 expm1 beta dawson digamma erfcx erfi exponent eta zeta gamma hankelh1 hankelh2 ceil floor round trunc iceil ifloor iround itrunc isfinite isnan lbeta lfact lgamma log log10 log1p log2 copysign max min significand sqrt hypot

Furthermore, Julia provides the `@vectorize_1arg` 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 to be the most general possible) and the name of the function to vectorize. Here is a simple example:

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

julia> @vectorize_1arg Number square
square (generic function with 4 methods)
```

```julia
julia> methods(square)
# 4 methods for generic function "square"

square{T<:Number}(x::AbstractArray{T<:Number,1}) at operators.jl:236
square{T<:Number}(x::AbstractArray{T<:Number,2}) at operators.jl:237
```
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  Float64 Array:
  0.848333  1.66714  1.3262
  1.26743   1.77988  1.13859
```

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  Float64 Array:
  0.848333  1.66714  1.3262
  1.26743   1.77988  1.13859
```

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 class. These operations are guaranteed to work correctly as a fallback for any specific array implementation.

The `Array{T,n}` type is a specific instance of `AbstractArray` where elements are stored in column-major order. `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 for `AbstractArray`. 

1.17. Multi-dimensional Arrays
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 indexing and memory allocation.

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

```julia
julia> a = rand(10,10)
10x10 Float64 Array:
  0.763921  0.884854  0.818783  0.519682  ...  0.860332  0.882295  0.420202
  0.190079  0.235315  0.0669517  0.020172  0.902405  0.0024219  0.24984
  0.823817  0.0285394  0.390379  0.202234  0.516727  0.247442  0.308572
  0.566851  0.622764  0.0683611  0.372167  0.280587  0.227102  0.145647
  0.151173  0.179177  0.0510514  0.615746  0.322073  0.245435  0.976068
  0.534307  0.493124  0.796481  0.0314695  ...  0.843201  0.53461  0.910584
  0.885078  0.891022  0.691548  0.547  0.727538  0.0218296  0.174351
  0.123628  0.833214  0.0224507  0.806369  0.80163  0.457005  0.226993
  0.362621  0.389317  0.702764  0.385856  0.155392  0.497805  0.430512
  0.504046  0.532631  0.477461  0.225632  0.919701  0.0453513  0.505329

julia> b = sub(a, 2:2:8,2:2:4)
4x2 SubArray of 10x10 Float64 Array:
  0.235315  0.020172
  0.622764  0.372167
  0.493124  0.0314695
  0.833214  0.806369

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

julia> q
4x2 Float64 Array:
 -0.200268  0.331205
 -0.530012  0.107555
 -0.41968  0.720129
 -0.709119  -0.600124

julia> r
2x2 Float64 Array:
 -1.175  -0.786311
  0.0  -0.414549
```

### 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:

```julia
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.

### Sparse matrix constructors

The simplest way to create sparse matrices are using 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
julia> spzeros(3,5)
3x5 sparse matrix with 0 nonzeros:

julia> speye(3,5)
3x5 sparse matrix with 3 nonzeros:
  [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
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 nonzeros:
  [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
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
julia> sparse(eye(5))
5x5 sparse matrix with 5 nonzeros:
  [1, 1] = 1.0
```
You can go in the other direction using the `dense` or the `full` function. The `issparse` function can be used to query if a matrix is sparse.

```julia
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 `find_nzs`, 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.

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

## 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.

<table>
<thead>
<tr>
<th>Cholesky</th>
<th>Cholesky factorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>CholeskyPivoted</td>
<td>Pivoted Cholesky factorization</td>
</tr>
<tr>
<td>LU</td>
<td>LU factorization</td>
</tr>
<tr>
<td>QR</td>
<td>Pivoted QR factorization</td>
</tr>
<tr>
<td>Hessenberg</td>
<td>Hessenberg decomposition</td>
</tr>
<tr>
<td>Eigen</td>
<td>Spectral decomposition</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular value decomposition</td>
</tr>
<tr>
<td>GeneralizedSVD</td>
<td>Generalized SVD</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Hermitian</th>
<th>Hermitian matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangular</td>
<td>Upper/lower triangular matrix</td>
</tr>
<tr>
<td>Tridiagonal</td>
<td>Tridiagonal matrix</td>
</tr>
<tr>
<td>SymTridiagonal</td>
<td>Symmetric tridiagonal matrix</td>
</tr>
<tr>
<td>Bidiagonal</td>
<td>Upper/lower bidiagonal matrix</td>
</tr>
<tr>
<td>Diagonal</td>
<td>Diagonal matrix</td>
</tr>
</tbody>
</table>

#### Elementary operations

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>\</th>
<th>Other functions with optimized methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermitian</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XY, inv, sqrtm, expm</td>
</tr>
<tr>
<td>Triangular</td>
<td></td>
<td>XY</td>
<td>XY</td>
<td>XY</td>
<td>inv, det</td>
</tr>
<tr>
<td>SymTridiagonal</td>
<td>X</td>
<td>X</td>
<td>XZ</td>
<td>XY</td>
<td>eigmax/min</td>
</tr>
<tr>
<td>Tridiagonal</td>
<td>X</td>
<td>X</td>
<td>XZ</td>
<td>XY</td>
<td></td>
</tr>
<tr>
<td>Bidiagonal</td>
<td>X</td>
<td>X</td>
<td>XZ</td>
<td>XY</td>
<td></td>
</tr>
<tr>
<td>Diagonal</td>
<td>X</td>
<td>X</td>
<td>XY</td>
<td>XY</td>
<td>inv, det, logdet, /</td>
</tr>
</tbody>
</table>

Legend:

- **X**: An optimized method for matrix-matrix operations is available
- **Y**: An optimized method for matrix-vector operations is available
- **Z**: An optimized method for matrix-scalar operations is available

#### Matrix factorizations

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>LAPACK</th>
<th>eig</th>
<th>eigvals</th>
<th>eigvecs</th>
<th>svd</th>
<th>svdvals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermitian</td>
<td>HE</td>
<td>ABC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangular</td>
<td>TR</td>
<td>ABC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SymTridiagonal</td>
<td>ST</td>
<td>A</td>
<td>ABC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tridiagonal</td>
<td>GT</td>
<td></td>
<td>AD</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Bidiagonal</td>
<td>BD</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagonal</td>
<td>DI</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.18. Linear algebra 113
Legend:

<table>
<thead>
<tr>
<th></th>
<th>An optimized method to find all the characteristic values and/or vectors is available</th>
<th>e.g. eigvals(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>An optimized method to find the $i^\text{th}$ through the $i^\text{th}$ characteristic values are available</td>
<td>eigvals(M, $i_l$, $i_h$)</td>
</tr>
<tr>
<td>B</td>
<td>An optimized method to find the characteristic values in the interval $[v_l, v_h]$ is available</td>
<td>eigvals(M, $v_l$, $v_h$)</td>
</tr>
<tr>
<td>C</td>
<td>An optimized method to find the characteristic vectors corresponding to the characteristic values $x=[x_1, x_2, \ldots]$ is available</td>
<td>eigvecs(M, $x$)</td>
</tr>
</tbody>
</table>

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
julia> write(STDOUT,"Hello World")
Hello World

julia> read(STDIN,Char)
'

Note that I pressed enter again so that Julia would read the newline. Now, as you can see from this example, the *write* method takes the data to write as its second argument, while the *read* method takes the type of the data to be read as the second argument. For example, to read a simply byte array, we could do:

```julia
julia> x = zeros(Uint8,4)
4-element Uint8 Array:
0x00
0x00
0x00
0x00

julia> read(STDIN,x)
abcd
4-element Uint8 Array:
0x61
0x62
0x63
0x64
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```julia
julia> readbytes(STDIN,4)
abcd
4-element Uint8 Array:
0x61
```

114 Chapter 1. The Julia Manual
or if we had wanted to read the entire line instead:

```
julia> readline(STDIN)
abcd
```

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

### 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 A simple TCP example

Let’s jump right in with a simple example involving Tcp Sockets. To do, let’s first create a simple server:

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

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:

```
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)
```
**1.19.4 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
julia> connect("google.com",80)
TcpSocket(open, 0 bytes waiting)
```

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

```julia
julia> Base.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 very relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the “ownership” of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia’s implementation of message passing is different from other environments such as MPI\(^3\). 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.

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

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

```julia
julia> fetch(r)
```

```julia
2x2 Float64 Array:
0.60401 0.501111
0.174572 0.157411
```

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

```julia
julia> fetch(s)
```

```julia
2x2 Float64 Array:
```

\(^3\) 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](http://www.mpi-forum.org/docs).
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.)

One important point is that 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
```
Processor 1 knew about the function `rand2`, but process 2 did not. To make your code available to all processes, the `require` function will automatically load a source file on all currently available processes:

```julia
julia> require("myfile")
```

In a cluster, the contents of the file (and any files loaded recursively) will be sent over the network. It is also useful to execute a statement on all processes. This can be done with the `@everywhere` macro:

```julia
julia> @everywhere id = myid()
```

```julia
julia> remotecall_fetch(2, () -> id)
2
```

@everywhere include("defs.jl")

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

```julia
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 processors 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.
- And a cluster spanning machines using the `--machinefile` option. This uses `ssh` to start the worker processes 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. See section on ClusterManagers.

### 1.20.1 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:

```julia
# 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 very 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.

### 1.20.2 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`:

```julia
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:

```julia
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

This example, as simple as it is, 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:

```julia
nheads = @parallel (+) for i=1:20000000
    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...
Julia Language Documentation, Release 0.2.0

iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```julia
a = zeros(100000)
@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 if 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:

```julia
a = randn(1000)
@parallel (+) for i=1:100000
    f(a[randi(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:

```julia
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.

### 1.20.3 Synchronization With Remote References

### 1.20.4 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:

```julia
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
        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.

1.20.5 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 processors to use, and how the data should be distributed:
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 `DArray`s 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 nth 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 processors.

distribute(a::Array) converts a local array to a distributed array.
localpart(a::DArray) obtains the locally-stored portion of a `DArray`.
myindexes(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 processor.

Indexing a `DArray` (square brackets) with ranges of indexes always creates a `SubArray`, not copying any data.

### 1.20.6 Constructing Distributed Arrays

The primitive `DArray` constructor has the following somewhat elaborate signature:

```julia
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 processor 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:

```julia
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.

### 1.20.7 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 processor needs the immediate neighbor cells of its local chunk. The following code accomplishes this:

```julia
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
```
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:

```julia
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 ? 1 :
                            nc == 2 ? old[i,j] :
                            0)
        end
    end
    new
end
```

### 1.20.8 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`:

```julia
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`:

```julia
addprocs(5, cman=LocalManager())
```

which specifies a number of processes to add and a `ClusterManager` to use for launching those processes.

## 1.21 Running External Programs

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

```julia
julia> 'echo hello'
'echo hello'
```

differs in a 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 actually running an external program:

```julia
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, the `readall` method can be used instead:

```julia
julia> a = readall('echo hello')
"hello\n"

julia> (chomp(a)) == "hello"
true
```

### 1.21.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use `§` for interpolation much as you would in a string literal (see `Strings`):

```julia
julia> file = "'/etc/passwd"
"/etc/passwd"

julia> 'sort $file'
'sort /etc/passwd'
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted to sort the contents of the file `/Volumes/External HD/data.csv`. Let’s try it:
How did the file name get quoted? Julia knows that \texttt{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 \texttt{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:

\begin{verbatim}
 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'"
\end{verbatim}

As you can see, the space in the \texttt{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):

\begin{verbatim}
 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'"
\end{verbatim}

If you interpolate an array as part of a shell word, Julia emulates the shell’s \{a,b,c\} argument generation:

\begin{verbatim}
 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'
\end{verbatim}

Moreover, if you interpolate multiple arrays into the same word, the shell’s Cartesian product generation behavior is emulated:

\begin{verbatim}
 julia> names = ["foo","bar","baz"]
 3-element ASCIIString Array:
    "foo"
    "bar"
    "baz"

 julia> exts = ["aux","log"]
 2-element ASCIIString Array:
    "aux"
    "log"
\end{verbatim}
Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
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'
```

### 1.21.2 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.
1.21.3 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
def run(cmd):
    print(cmd)
run('echo 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
def run(cmd1, cmd2):
    print(cmd1 |> cmd2)
run('echo hello' |> 'sort')
```

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
def run(cmd1, cmd2, cmd3):
    print(cmd1 |> cmd2 |> cmd3)
run('cut -d: -f3 /etc/passwd' |> 'sort -n' |> 'tail -n5')
```

This prints the highest five user IDs on a UNIX system. The `cut`, `sort` and `tail` commands are all spawned as immediate children of the current `julia` process, with no intervening shell process. Julia itself does the work to setup pipes and connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

```julia
def run(cmd1, cmd2):
    print(cmd1 & cmd2)
run('echo hello' & 'echo world')
```

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
def run(cmd1, cmd2, cmd3):
    print(cmd1 |> cmd2 |> cmd3)
run('echo world' & 'echo hello' |> 'sort')
```

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
def prefixer(prefix, sleep):
    return 'perl -nle "$|=1; print "$prefix", $_; sleep "$sleep";"'

def run(cmd1, cmd2, cmd3):
    print(cmd1 |> cmd2 |> cmd3)
run('perl -le "$|=1; for(0..9){ print; sleep 1 };"' |> prefixer("A",2) & prefixer("B",2))
```

Julia lets you pipe the output from both of these processes to another program:
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
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.

### 1.22 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 `call` 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.

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`:

```julia
function getenv(var::String)
    val = ccall( (:getenv, "libc"),
                 Ptr{Uint8}, (Ptr{Uint8},), bytestring(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:

```julia
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.

When calling a Fortran function, all inputs must be passed by reference.

A prefix `&` is used to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself. The following example computes a dot product using a BLAS function.

```julia
function compute_dot(DX::Vector, DY::Vector)
    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. However, it will never cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Since this `&` is not a real address operator, it may be used with any syntax, such as `&0` or `&f(x)`.

Note that no C header files are used anywhere in the process. Currently, it is not possible to pass structs and other non-primitive types from Julia to C libraries. However, C functions that generate and use opaque structs types by passing around pointers to them can return such values to Julia as `Ptr{Void}`, which can then be passed to other C functions as `Ptr{Void}`. 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.

### 1.22.1 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:

```julia
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` argument, it is “converted” simply by taking the address of the first element. This is done in order to avoid copying arrays unnecessarily, and to tolerate the slight mismatches in pointer types that are often encountered in C APIs (for example, passing a `Float64` array to a function that operates on uninterpreted bytes).

Therefore, if an Array contains data in the wrong format, it will have to be explicitly converted using a call such as `int32(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:**

<table>
<thead>
<tr>
<th>C type</th>
<th>Julia type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code> (8 bits)</td>
<td>Cbool</td>
</tr>
<tr>
<td><code>signed char</code></td>
<td>Cuchar</td>
</tr>
<tr>
<td><code>unsigned char</code></td>
<td>CUchar</td>
</tr>
<tr>
<td><code>short</code></td>
<td>Cshort</td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td>CUshort</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Cint</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>CUint</td>
</tr>
<tr>
<td><code>long long</code></td>
<td>Cllonglong</td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td>Cullonglong</td>
</tr>
<tr>
<td><code>float</code></td>
<td>Cfloat</td>
</tr>
<tr>
<td><code>double</code></td>
<td>Cdouble</td>
</tr>
<tr>
<td><code>ptrdiff_t</code></td>
<td>Cptrdiff_t</td>
</tr>
<tr>
<td><code>size_t</code></td>
<td>Csize_t</td>
</tr>
<tr>
<td><code>complex float</code></td>
<td>CComplex_float</td>
</tr>
<tr>
<td><code>complex double</code></td>
<td>CComplex_double</td>
</tr>
<tr>
<td><code>void</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>void*</code></td>
<td>Ptr{Void}</td>
</tr>
<tr>
<td><code>char*</code> (or char[], e.g. a string)</td>
<td>Ptr{Uint8}</td>
</tr>
<tr>
<td><code>char**</code> (or *char[])</td>
<td>Ptr{Ptr{Uint8}}</td>
</tr>
<tr>
<td><code>struct T*</code> (where T represents an appropriately defined bits type)</td>
<td>Ptr{T} (call using &amp;variable_name in the parameter list)</td>
</tr>
<tr>
<td><code>struct T</code> (where T represents an appropriately defined bits type)</td>
<td>T (call using &amp;variable_name in the parameter list)</td>
</tr>
<tr>
<td><code>jl_value_t*</code> (any Julia Type)</td>
<td>Ptr{Any}</td>
</tr>
</tbody>
</table>

**Note:** the `bool` type is only defined by C++, where it is 8 bits wide. In C, however, `int` is often used for boolean values. Since `int` is 32-bits wide (on all supported systems), there is some potential for confusion here.

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:**
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:

```c
int main(int argc, char **argv);
```

can be called via the following Julia code:

```julia
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{Uint8}}), length(argv), argv)
```

### 1.22.2 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 bitstypes 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.

### 1.22.3 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.

### 1.22.4 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:

```julia
@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.

### 1.22.5 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.

### 1.22.6 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):

```julia
hn = Array(Uint8, 256)
err=ccall(:gethostname, stdcall, Int32, (Ptr{Uint8}, Uint32), hn, length(hn))
```

For more information, please see the LLVM Language Reference.

### 1.22.7 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```julia
julia> cglobal((:errno,:libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store`. 
1.22.8 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:

```c
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 `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`:

```julia
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`:

```julia
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:

```julia
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).

1.22.9 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

1.22.10 C++

Limited support for C++ is provided by the Cpp and Clang packages.

1.22.11 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:
Julia Language Documentation, Release 0.2.0

@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:

```julia
ccall( (@windows? :_fopen : :fopen), ...)
```

Complex blocks:

```julia
@linux? (begin
    some_complicated_thing(a)
    end
    : begin
        some_different_thing(a)
        end
    )
```

Chaining (parentheses optional, but recommended for readability):

```julia
```

### 1.23 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 [available-packages](#). All package manager commands are found in the `Pkg` module, included in Julia’s Base install.

#### 1.23.1 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
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:
1.23.2 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/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/REQUIRE file:

```
$ cat ~/.julia/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/REQUIRE file by hand and then running Pkg.resolve() yourself:

```
$ echo UTF16 >> ~/.julia/REQUIRE
```

```
    julia> Pkg.resolve()
    INFO: Cloning cache of UTF16 from git://github.com/nolta/UTF16.jl.git
    INFO: Installing UTF16 v0.2.0
```
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. 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.
```

1.23.3 Installing Unregistered Packages

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.
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` 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.

### 1.23.4 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
```

The first step of updating packages is to pull new changes to `~/.julia/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/REQUIRE` and the requirement of fixed packages are used to determine what should be installed.

### 1.23.5 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:
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.

### 1.23.6 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.

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.

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/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 .travis.yml
INFO: Committing FooBar generated files

This creates the directory ~/.julia/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/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
    github: true
    travis: true

Julia Version 0.2.0-rc1+23 [2039ec61a5]
.travis.yml | 13 ++++++++++++++
LICENSE.md | 23 ++++++++++++++++++++++
README.md | 3 +++
src/FooBar.jl | 5 +++
4 files changed, 44 insertions(+)

At the moment, the package manager knows about the MIT “Expat” License, indicated by "MIT", and the Simplified BSD License, indicated by "BSD". 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 two and then modify the ~/.julia/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.

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 out\textsuperscript{5}. 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:

\begin{verbatim}
git://github.com/StefanKarpinski/FooBar.jl.git
\end{verbatim}

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 \texttt{Pkg.clone} to install the package and try it out:

\begin{verbatim}
julia> Pkg.clone("git://github.com/StefanKarpinski/FooBar.jl.git")
INFO: Cloning FooBar from git://github.com/StefanKarpinski/FooBar.jl.git
Cloning into 'FooBar'...
remote: Counting objects: 22, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 22 (delta 7), reused 21 (delta 6)
Receiving objects: 100% (22/22), done.
Resolving deltas: 100% (7/7), done.
\end{verbatim}

Once you’ve decided that \texttt{FooBar} is ready to be registered as an official package, you can add it to your local copy of \texttt{METADATA} using \texttt{Pkg.register}:

\begin{verbatim}
julia> Pkg.register("FooBar")
INFO: Registering FooBar at git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Committing METADATA for FooBar
\end{verbatim}

This creates a commit in the \texttt{~/.julia/METADATA} repo:

\begin{verbatim}
$ cd ~/.julia/METADATA && git show

commit 9f71f4becb05cadacb983c54a72e674e5c019d
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
@@ -0,0 +1 @@
+git://github.com/StefanKarpinski/FooBar.jl.git
\end{verbatim}

This commit is only locally visible, however. In order to make it visible to the world, you need to merge your local \texttt{METADATA} upstream into the official repo. If you have push access to that repository (which we give to all package maintainers), then you can do so easily with the \texttt{Pkg.publish()} command, which publishes your local metadata changes. If you don’t have push access to \texttt{METADATA}, you’ll have to make a pull request on GitHub, which is not difficult.

Once the package URL for \texttt{FooBar} is registered in the official \texttt{METADATA} repo, people know where to clone the package from, but there still aren’t any registered versions available. This means that \texttt{Pkg.add("FooBar")} won’t work yet since it only installs official versions. People can, however, clone the package with just \texttt{Pkg.clone("FooBar")} without having to specify a URL for it. Moreover, when they run \texttt{Pkg.update()}, they will get the latest version of \texttt{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.

Once you are ready to make an official version your package, you can tag and register it with the \texttt{Pkg.tag} command:

\begin{verbatim}
\end{verbatim}

\begin{footnote}{Installing and using GitHub’s “hub” tool is highly recommended. It allows you to do things like run \texttt{hub create} in the package repo and have it automatically created via GitHub’s API.}

1.23. Packages 143
This tags v0.0.0 in the FooBar repo:

```
$ cd ~/.julia/FooBar && git tag
v0.0.0
```

It also creates a new version entry in your local METADATA repo for FooBar:

```
$ cd ~/.julia/FooBar && git show
commit de77e4dc0689b12c5e8b574aef7f70e8b311b0e
Author: Stefan Karpinski <stefan@karpinski.org>
Date: Wed Oct 16 23:06:18 2013 -0400

    Tag FooBar v0.0.0

diff --git a/FooBar/versions/0.0.0/sha1 b/FooBar/versions/0.0.0/sha1
new file mode 100644
index 0000000..c1cb1c1
--- /dev/null
+++ b/FooBar/versions/0.0.0/sha1
@@ -0,0 +1 @@
+84b8e266dae6de30ab9703150b3bf771ec7b6285
```

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.

These changes to METADATA aren’t available to anyone else until they’ve been included upstream. If you have push access to the official METADATA repo, you can 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 pushes METADATA to the origin. If you don’t have push access to METADATA, you’ll have to open a pull request for the last bit, although we’re planning on automatically opening pull requests for you in the future.

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. 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. 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.

## 1.23.7 Requirements

The ~/.julia/REQUIRE file and REQUIRE files inside of packages use a simple line-based format to express what ranges of package versions are needed. 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. 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 1.x version will do, you will want to write:

Distributions 0.1 0.2-

The 0.2−“pseudo-version” is less than all real version numbers that start with 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.

### 1.24 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.

#### 1.24.1 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:
Uses of non-constant globals can be optimized by annotating their types at the point of use:

```julia
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.

### 1.24.2 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```julia
a = Real[]  # 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:

```julia
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*.

### 1.24.3 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:

```julia
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:

```julia
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.

### 1.24.4 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:

```julia
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(x,x)))
    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:

```julia
norm(A::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = max(svd(A)[2])
```

### 1.24.5 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:

```julia
pos(x) = x < 0 ? 0 : x
```
Although this seems innocent enough, the problem is that 0 is an integer (of type \texttt{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:

\[
\text{pos}(x) = x < 0 \, ? \, \text{zero}(x) : x
\]

There is also a \texttt{one} function, and a more general \texttt{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.

### 1.24.6 Avoid changing the type of a variable

An analogous “type-stability” problem exists for variables used repeatedly within a function:

```julia
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 the \(/\) 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 \(x\): \(x::\text{Float64} = 1\)
- Use an explicit conversion: \(x = \text{one}(T)\)

### 1.24.7 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:

```julia
function strange_twos(n)
    a = Array(randbool() ? \texttt{Int64} : \texttt{Float64}, n)
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

This should be written as:

```julia
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
    end
end

function strange_twos(n)
    a = Array(randbool() ? \texttt{Int64} : \texttt{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 \texttt{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 \texttt{hvcat\_fill} in \texttt{abstractarray.jl}, or the \texttt{fill!} function, which we could have used instead of writing our own \texttt{fill\_twos!}.

Functions like \texttt{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.

### 1.24.8 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.

### 1.24.9 Tweaks

These are some minor points that might help in tight inner loops.

- Use \texttt{size(A,n)} when possible instead of \texttt{size(A)} or \texttt{size(A)[n]}.
- Avoid unnecessary arrays. For example, instead of \texttt{sum([x,y,z])} use \texttt{x+y+z}.
- Use * instead of raising to small integer powers, for example \texttt{x\*x\*x} instead of \texttt{x\^3}.
- Use \texttt{abs2(z)} instead of \texttt{abs(z)^2} for complex \( z \). In general, try to rewrite code to use \texttt{abs2} instead of \texttt{abs} for complex arguments.
- Use \texttt{div(x,y)} for truncating division of integers instead of \texttt{ trunc(x/y), and fld(x,y) instead of floor(x/y)}.

### 1.25 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.

#### 1.25.1 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 \texttt{pi}).

#### 1.25.2 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:

```julia
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 the one function (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.

### 1.25.3 Handle excess argument diversity in the caller

Instead of:

```julia
function foo(x, y)
    x = int(x); y = int(y)
    ...
end
foo(x, y)
```

use:

```julia
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.

### 1.25.4 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.

### 1.25.5 Avoid strange type Unions

Types such as Union(Function,String) are often a sign that some design could be cleaner.

### 1.25.6 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.

### 1.25.7 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.

### 1.25.8 Avoid underscores in names

If a function name requires multiple words, it might represent more than one concept. It is better to keep identifier names concise.

### 1.25.9 Don’t overuse try-catch

It is better to avoid errors than to rely on catching them.
1.25.10 Don't parenthesize conditions

Julia doesn’t require parens around conditions in if and while. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

1.25.11 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.

1.25.12 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. 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.

1.25.13 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.

1.25.14 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.
1.25.15 Don’t expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```julia
type NativeType
    p::Ptr{Uint8}
    ...
end
```

don’t write definitions like the following:

```julia
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.

1.25.16 Don’t overload methods of base container types

It is possible to write definitions like the following:

```julia
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.

1.25.17 Be careful with type equality

You generally want to use `isa` and `<:` (subtype) 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.

1.25.18 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)`.

1.26 Frequently Asked Questions

1.26.1 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:

```julia
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)
...
```

### 1.26.2 Type declarations and constructors

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
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType

julia> typeof(c)
MyAmbiguousType
```

`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:

```
type MyType{T<:FloatingPoint}
    a::T
end
```

This is a better choice than

```
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}

julia> typeof(t)
MyStillAmbiguousType
```

The type of field `a` can be readily determined from the type of `m`, but not from the type of `t`. Indeed, in `t` it’s possible to change the type of 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

type MyAmbiguousContainer{T}
    a::AbstractVector{T}
end
```

For example:

```julia
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer(Rangel{Int64})

julia> c = MySimpleContainer([1:3]);

julia> typeof(c)
MySimpleContainer{Array{Int64,1}}

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3]);

julia> typeof(b)
MyAmbiguousContainer{Int64}
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where you might wish that your code could do different things depending on the element type of `a`. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:
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:

```julia
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 `Range1{T}` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `MyContainer`:

```julia
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,Range1{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:

```julia
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))

1.26. Frequently Asked Questions 157
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
julia> b = MyContainer{Int64, Rangel{Float64}}(1.3:5);
```

To prevent this, we can add an inner constructor:

```julia
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
julia> b = MyBetterContainer(1.3:5);
```

The inner constructor requires that the element type of $A$ be $T$.

### 1.26.3 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.
1.26.4 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 slower 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.

1.26.5 Developing Julia

How do I debug julia’s C code? (running the julia REPL from within a debugger like gdb)

First, you should build the debug version of julia with `make debug`. Below, lines starting with `(gdb)` mean things you should type at the gdb prompt.

From the shell

The main challenge is that Julia and gdb each need to have their own terminal, to allow you to interact with them both. One approach is to use gdb’s `attach` functionality to debug an already-running julia session. However, on many systems you’ll need root access to get this to work. What follows is a method that can be implemented with just user-level permissions.

The first time you do this, you’ll need to define a script, here called `oterm`, containing the following lines:

```bash
ps
sleep 600000
```

Make it executable with `chmod +x oterm`.

Now:

- From a shell (called shell 1), type `xterm -e oterm &`. You’ll see a new window pop up; this will be called terminal 2.
- From within shell 1, `gdb julia-debug-basic`. You can find this executable within `julia/usr/bin`.
- From within shell 1, `(gdb) tty /dev/pts/#` where `#` is the number shown after `pts/` in terminal 2.
- From within shell 1, `(gdb) run`
• From within terminal 2, issue any preparatory commands in Julia that you need to get to the step you want to debug
• From within shell 1, hit Ctrl-C
• From within shell 1, insert your breakpoint, e.g., (gdb) b codegen.cpp:2244
• From within shell 1, (gdb) c to resume execution of julia
• From within terminal 2, issue the command that you want to debug. Shell 1 will stop at your breakpoint.

**Within emacs**

• M-x gdb, then enter julia-debug-basic (this is easiest from within julia/usr/bin, or you can specify the full path)
• (gdb) run
• Now you’ll see the Julia prompt. Run any commands in Julia you need to get to the step you want to debug.
• Under emacs’ “Signals” menu choose BREAK—this will return you to the (gdb) prompt
• Set a breakpoint, e.g., (gdb) b codegen.cpp:2244
• Go back to the Julia prompt via (gdb) c
• Execute the Julia command you want to see running.

### 1.27 Noteworthy Differences from other Languages

#### 1.27.1 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.
• The imaginary unit sqrt(-1) is represented in julia with im.
• Multiple values are returned and assigned with parentheses, return (a, b) and (a, b) = f(x).
• Julia has 1-dimensional arrays. Column vectors are of size N, not Nx1. 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 \texttt{sum}, \texttt{prod}, and \texttt{max} are performed over every element of an array when called with a single argument as in \texttt{sum(A)}.

• Functions such as \texttt{sort} that operate column-wise by default (\texttt{sort(A)} is equivalent to \texttt{sort(A,1)}) do not have special behavior for 1xN arrays; the argument is returned unmodified since it still performs \texttt{sort(A,1)}. To sort a 1xN matrix like a vector, use \texttt{sort(A,2)}.

• Parentheses must be used to call a function with zero arguments, as in \texttt{tic()} and \texttt{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 \texttt{println} can be used to print a value followed by a newline.

• If \texttt{A} and \texttt{B} are arrays, \texttt{A == B} doesn’t return an array of booleans. Use \texttt{A .== B} instead. Likewise for the other boolean operators, \texttt{<}, \texttt{>, !}, etc.

• The elements of a collection can be passed as arguments to a function using \ldots, as in \texttt{xs=[1,2]; f(xs...)}.

• Julia’s \texttt{svd} returns singular values as a vector instead of as a full diagonal matrix.

• In Julia, \ldots is not used to continue lines of code.

• The variable \texttt{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.

1.27.2 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 \texttt{=} for assignment. Julia does not provide any operator like \texttt{<-} or \texttt{<<-}.

• Julia constructs vectors using brackets. Julia’s \texttt{[1, 2, 3]} is the equivalent of R’s \texttt{c(1, 2, 3)}.

• Julia’s matrix operations are more like traditional mathematical notation than R’s. If \texttt{A} and \texttt{B} are matrices, then \texttt{A * B} defines a matrix multiplication in Julia equivalent to R’s \texttt{A %*% B}. In R, this same notation would perform an elementwise Hadamard product. To get the elementwise multiplication operation, you need to write \texttt{A .* B} in Julia.

• Julia performs matrix transposition using the ‘ operator. Julia’s \texttt{A}’ is therefore equivalent to R’s \texttt{t(A)}.

• Julia does not require parenses when writing if statements or for loops: use \texttt{for i in [1, 2, 3]} instead of \texttt{for (i in c(1, 2, 3))} and if \texttt{i == 1} instead of if \texttt{(i == 1)}.

• Julia does not treat the numbers \texttt{0} and \texttt{1} as Booleans. You cannot write \texttt{if (1)} in Julia, because if statements accept only booleans. Instead, you can write \texttt{if true}.

• Julia does not provide \texttt{nrow} and \texttt{ncol}. Instead, use \texttt{size(M, 1)} for \texttt{nrow(M)} and \texttt{size(M, 2)} for \texttt{ncol(M)}.

• Julia’s SVD is not thinned by default, unlike R. To get results like R’s, you will often want to call \texttt{svd(X, true)} on a matrix \texttt{X}.

• Julia is very careful to distinguish scalars, vectors and matrices. In R, \texttt{1} and \texttt{c(1)} are the same. In Julia, they can not be used interchangeably. One potentially confusing result of this is that \texttt{x’ * y} for vectors \texttt{x} and \texttt{y} is a 1-element vector, not a scalar. To get a scalar, use \texttt{dot(x, y)}.

• Julia’s \texttt{diag()} and \texttt{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 \texttt{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.
- Formulas for GLM’s must be escaped: use `(y ~ x)` instead of `y ~ x`.

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]`.
- 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.

### 1.27.3 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.
CHAPTER 2

The Julia Standard Library

2.1 Built-ins

2.1.1 Introduction

The Julia standard library contains a range of functions and macros appropriate for performing scientific and numerical computing, but as broad as 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, 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 ModuleName 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

Calls exit(0).

atexit (f)

Register a zero-argument function to be called at exit.

isinteractive ()

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.
**Julia Language Documentation, Release 0.2.0**

**edit (function[, types ])**
Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit.

**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.

**clipboard(x)**
Send a printed form of x to the operating system clipboard (“copy”).

**clipboard() → String**
Return 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 the system-wide LOAD_PATH 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.

**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, args...)**
Show which method of f will be called for the given arguments.

**@which ()**
Evaluates the arguments to the function call, determines their types, and calls the which function on the resulting expression.

**methods (f)**
Show all methods of f with their argument types.

**methodswith (typ[, showparents ])**
Show all methods with an argument of type typ. If optional showparents is true, also show 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.

### 2.1.3 All Objects

**is** (*x*, *y*)

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 `equal`. The `===` operator is an alias for this function.

**isa** (*x*, *type*)

Determine whether *x* is of the given type.

**isequal** (*x*, *y*)

True if and only if *x* and *y* have the same contents. Loosely speaking, this means *x* and *y* would look the same when printed. This is the default comparison function used by hash tables (Dict). New types with a notion of equality should implement this function, except for numbers, which should implement `==` instead. However, numeric types with special values might need to implement `isequal` as well. For example, floating point NaN values are not `==`, but are all equivalent in the sense of `isequal`. Numbers of different types are considered unequal. Mutable containers should generally implement `isequal` by calling `isequal` recursively on all contents.

**isless** (*x*, *y*)

Test whether *x* is less than *y*. Provides a total order consistent with `isequal`. 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 that can be ordered 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*)

Compute an integer hash code such that `isequal(x, y)` implies `hash(x) == hash(y)`.

**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 \((object, index \mid symbol)\)
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.

convert \((\text{type}, x)\)
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 \(x\).

identity \((x)\)
The identity function. Returns its argument.

### 2.1.4 Types

super \((T::DataType)\)
Return the supertype of `DataType T`.

issubtype \((\text{type1, type2})\)
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 \((\text{type})\)
The lowest value representable by the given (real) numeric type.

typemax \((\text{type})\)
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. The only types that are sensible arguments are **Float32** and **Float64**. 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 which 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)...]
structinfo(Stat)
```

**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.

**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}**.

**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

**method_exists** \((f, \text{tuple}) \rightarrow \text{Bool}\)

Determine whether the given generic function has a method matching the given tuple of argument types.

**Example:** `method_exists(length, (Array,)) = true`

**applicable** \((f, \text{args}...)\)

Determine whether the given generic function has a method applicable to the given arguments.

**invoke** \((f, \text{types}..., \text{args}...)\)

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).

\(\Rightarrow (x, f)\)

Applies a function to the preceding argument which allows for easy function chaining.

**Example:** \([1:5] \Rightarrow x\rightarrow x.^2 \Rightarrow \text{sum} \Rightarrow \text{inv}\)

### 2.1.6 Syntax

**eval** \((expr::Expr)\)

Evaluate an expression and return the value.

**@eval** ()

Evaluate an expression and return the value.

**evalfile** \((\text{path}::\text{String})\)

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** \((e::\text{ANY})\)

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 Macros section of the Metaprogramming chapter of the manual for more details and examples.

**gensym** \([\text{tag}]\)

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** \((\text{str}, \text{start}; \text{greedy=true, raise=false})\)

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 (default is 1). 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 token. If raise is true (default), parse errors will raise an error; otherwise, parse will return the error as an expression object.
## 2.1.7 Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`. The general `for` loop:

```julia
for i = I
    # body
end
```

is translated to:

```julia
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:
  
  \[
  \text{zip(zip(a...)...)}\ldots = [a\ldots].
  \]

- `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.

Fully implemented by: `Range, Rangel, NDRange, Tuple, Real, AbstractArray, IntSet, ObjectIdDict, Dict, WeakKeyDict, EachLine, String, Set, Task`.

## 2.1.8 General Collections

- `isempty(collection) → Bool`
  Determine whether a collection is empty (has no elements).

- `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.

  **Example:** `endof([1,2,4]) = 3`

Fully implemented by: `Range, Rangel, Tuple, Number, AbstractArray, IntSet, Dict, WeakKeyDict, String, Set`.

---

## 2.1. Built-ins

---
2.1.9 Iterable Collections

in(item, collection) → Bool
Determine whether an item is in the given collection, in the sense that it is isequal to one of the values generated by iterating over the collection.

elementtype(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)
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.

reduce(op, v0, itr)
Reduce the given collection with the given operator, i.e. accumulate v = op(v, elt) for each element, where v starts as v0. Reductions for certain commonly-used operators are available in a more convenient 1-argument form: maximum(itr), minimum(itr), sum(itr), prod(itr), any(itr), all(itr).

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.

maximum(itr)
Returns the largest element in a collection

maximum(A, dims)
Compute the maximum value of an array over the given dimensions

minimum(itr)
Returns the smallest element in a collection

minimum(A, dims)
Compute the minimum value of an array over the given dimensions

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

findmin(itr) -> (x, index)
Returns the minimum element and its index

sum(itr)
Returns the sum of all elements in a collection

sum(A, dims)
Sum elements of an array over the given dimensions.

sum(f, itr)
Sum the results of calling function f on each element of itr.
prod(itr)
   Returns the product of all elements of a collection

prod(A, dims)
   Multiply elements of an array over the given dimensions.

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.

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.

count(p, itr) → Integer
   Count the number of elements in itr for which predicate p is true.

any(p, itr) → Bool
   Determine whether any element of itr satisfies the given predicate.

all(p, itr) → Bool
   Determine whether all elements of itr satisfy the given predicate.

map(f, c) → collection
   Transform collection c by applying f to each element.

   Example: map((x) -> x * 2, [1, 2, 3]) = [2, 4, 6]

map!(function, collection)
   In-place version of map().

mapreduce(f, op, itr)
   Applies function f to each element in itr and then reduces the result using the binary function op.

   Example: mapreduce(x->x^2, +, [1:3]) == 1 + 4 + 9 == 14

   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.

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)
   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).

2.1.10 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, Rangel, Tuple.

2.1.11 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}.

Dict ()
Dict{K,V} () constructs a hashtable 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)
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.

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

sizeihint(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.

2.1.12 Set-Like Collections

add!(collection, key)
Add an element to a set-like collection.

Set(x...)
Construct a Set with the given elements. Should be used instead of IntSet for sparse integer sets, or for sets of arbitrary objects.

IntSet(i...)
Construct a sorted set of the given integers. Implemented as a bit string, and therefore designed for dense integer sets. If the set will be sparse (for example holding a single very large integer), use Set instead.

union(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...)
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.

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**

True if $A \subseteq S$ (A is a subset of or equal to S)

Fully implemented by: IntSet, Set.

Partially implemented by: Array.

### 2.1.13 Dequeues

**push! (collection, item) → collection**

Insert an item at the end of a collection.

**pop! (collection) → item**

Remove the last item in a collection and return it.

**unshift! (collection, item) → collection**

Insert an item 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.

**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.

**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.

**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. append!([1],[2,3]) => [1,2,3]

**prepend! (collection, items) → collection**

Insert the elements of items to the beginning of a collection. prepend!([3],[1,2]) => [1,2,3]

Fully implemented by: Vector (aka 1-d Array), BitVector (aka 1-d BitArray).

### 2.1.14 Strings

**length (s)**

The number of characters in string s.

**sizeof (s::String)**

The number of bytes in string s.
* (s, t)
  Concatenate strings.
  
  Example: "Hello " * "world" == "Hello world"

^ (s, n)
  Repeat string s n times.
  
  Example: "Julia " ^3 == "Julia Julia Julia "

string(xs...)  
Create a string from any values using the print function.

repr(x)  
Create a string from any value using the show function.

bytestring (::Ptr{Uint8})
  Create a string from the address of a C (0-terminated) string. A copy is made; the ptr can be safely freed.

bytestring(s)
  Convert a string to a contiguous byte array representation appropriate for passing it to C functions.

ascii (::Array{Uint8, 1})
  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, 1})
  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).

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.

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

eachmatch(r::Regex, s::String[, overlap::Bool=false ])
  Search for all matches of a the regular expression r in s and return a iterator over the matches. If overlap is true, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

matchall(r::Regex, s::String[, overlap::Bool=false ])
  Return a vector of the matching subtrings from eachmatch.

lpad(string, n, p)
  Make a string at least n characters long by padding on the left with copies of p.

rpad(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 speci-
ifies 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 un-
matched.

search(string, 'c') = index such that string[index] == 'c', or 0 if un-
matched.

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 strings 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 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 a string chars is provided, instead
remove characters contained in that string.

lstrip(string[, chars])
Return string with any leading whitespace removed. If a string chars is provided, instead remove characters
contained in that string.

rstrip(string[, chars])
Return string with any trailing whitespace removed. If a string chars is provided, instead remove characters
contained in that string.

beginswith(string, prefix)
Returns true if string starts with prefix.

endswith(string, suffix)
Returns true if string ends with suffix.

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)  
    Join an array of strings into a single string, inserting the given delimiter between adjacent strings.

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 endof(str)+1 at the end of the string.

prevind (str, i)  
    Get the previous valid string index before i. Returns 0 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))  
    Tests whether a character is alphanumeric, or whether this is true for all elements of a string.

isalpha (c::Union(Char, String))  
    Tests whether a character is alphabetic, or whether this is true for all elements of a string.

isascii (c::Union(Char, String))  
    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))  
    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))  
    Tests whether a character is a control character, or whether this is true for all elements of a string.

isdigit (c::Union(Char, String))  
    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))  
    Tests whether a character is printable, and not a space, or whether this is true for all elements of a string.
islower \((c::Union(Char, String))\)
Tests whether a character is a lowercase letter, or whether this is true for all elements of a string.

isprint \((c::Union(Char, String))\)
Tests whether a character is printable, including space, or whether this is true for all elements of a string.

ispunct \((c::Union(Char, String))\)
Tests whether a character is printable, and not a space or alphanumeric, or whether this is true for all elements of a string.

isspace \((c::Union(Char, String))\)
Tests whether a character is any whitespace character, or whether this is true for all elements of a string.

isupper \((c::Union(Char, String))\)
Tests whether a character is an uppercase letter, or whether this is true for all elements of a string.

isxdigit \((c::Union(Char, String))\)
Tests whether a character is a valid hexadecimal digit, or whether this is true for all elements of a string.

symbol \((str)\)
Convert a string to a Symbol.

escape_string \((str::String) \rightarrow String\)
General escaping of traditional C and Unicode escape sequences. See print_escaped() for more general escaping.

unescape_string \((s::String) \rightarrow String\)
General unescaping of traditional C and Unicode escape sequences. Reverse of escape_string(). See also print_unescaped().

2.1.15 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[\[, read, write, create, truncate, append \]]) \rightarrow 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[\[, mode \]]) \rightarrow 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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>read</td>
</tr>
<tr>
<td>r+</td>
<td>read, write</td>
</tr>
<tr>
<td>w</td>
<td>write, create, truncate</td>
</tr>
<tr>
<td>w+</td>
<td>read, write, create, truncate</td>
</tr>
<tr>
<td>a</td>
<td>write, create, append</td>
</tr>
<tr>
<td>a+</td>
<td>read, write, create, append</td>
</tr>
</tbody>
</table>

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.

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.

eof\((stream)\)
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\((stream)\)
Determine whether a stream is read-only.

isopen\((stream)\)
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 filename 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.

stat (file)
Returns a structure whose fields contain information about the file. The fields of the structure are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>The size (in bytes) of the file</td>
</tr>
<tr>
<td>device</td>
<td>ID of the device that contains the file</td>
</tr>
<tr>
<td>inode</td>
<td>The inode number of the file</td>
</tr>
<tr>
<td>mode</td>
<td>The protection mode of the file</td>
</tr>
<tr>
<td>nlink</td>
<td>The number of hard links to the file</td>
</tr>
<tr>
<td>uid</td>
<td>The user id of the owner of the file</td>
</tr>
<tr>
<td>gid</td>
<td>The group id of the file owner</td>
</tr>
<tr>
<td>rdev</td>
<td>If this file refers to a device, the ID of the device it refers to</td>
</tr>
<tr>
<td>blksize</td>
<td>The file-system preffered block size for the file</td>
</tr>
<tr>
<td>blocks</td>
<td>The number of such blocks allocated</td>
</tr>
<tr>
<td>mtime</td>
<td>Unix timestamp of when the file was last modified</td>
</tr>
<tr>
<td>ctime</td>
<td>Unix timestamp of when the file was created</td>
</tr>
</tbody>
</table>
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.

cftime (file)
Equivalent to stat(file).ctime

mtime (file)
Equivalent to stat(file).mtime

filename (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

<table>
<thead>
<tr>
<th>Number</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Execute</td>
</tr>
<tr>
<td>02</td>
<td>Write</td>
</tr>
<tr>
<td>04</td>
<td>Read</td>
</tr>
</tbody>
</table>

For allowed arguments, see the stat method.

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 dest.

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)
Delete the file at the given path. Note that this does not work on directories.

touch (path::String)
Update the last-modified timestamp on a file to the current time.

2.1.16 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.

2.1.17 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 \(\text{string}(\text{typeof}(x))\). 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 \((\text{color::Symbol}, \text{io}, \text{strings...})\)
Print strings in a color specified as a symbol, for example :red or :blue.

info \((\text{msg})\)
Display an informational message.

warn \((\text{msg})\)
Display a warning.

@printf \([\text{io::IOStream}], \text{“Fmt”}, \text{args...}\)
Print arg(s) using C printf() style format specification string. Optionally, an IOStream may be passed as the first argument to redirect output.

@sprintf \((\text{“Fmt”}, \text{args...})\)
Return @printf formatted output as string.

sprint \((\text{f::Function}, \text{args...})\)
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 \((\text{io}, \text{e})\)
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 \((\text{stream})\)
Read the entire contents of an I/O stream as a string.

readline \((\text{stream})\)
Read a single line of text, including a trailing newline character (if one is reached before the end of the input).

readuntil \((\text{stream}, \text{delim})\)
Read a string, up to and including the given delimiter byte.

readlines \((\text{stream})\)
Read all lines as an array.

eachline \((\text{stream})\)
Create an iterable object that will yield each line from a stream.

readdlm \((\text{source}, \text{delim::Char}; \text{has_header=false}, \text{use_mmap=false}, \text{ignore_invalid_chars=false})\)
Read a matrix from the source where each line gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped filed can be used by passing the byte array representation of the mapped segment as source.

If \(\text{has_header}\) is \text{true} the first row of data would be read as headers and the tuple \((\text{data_cells}, \text{header_cells})\) is returned instead of only \text{data_cells}. 
If `use_mmap` is true the file specified by `source` is memory mapped for potential speedups.

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 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...)```
Read a matrix from the source with a given element type. 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.

```writedlm(filename, array, delim::Char)```
Write an array to a text file using the given delimeter (defaults to comma).

```readcsv(source, [T::Type]; options...)```
Equivalent to `readdlm` with `delim` set to comma.

```writecsv(filename, array)```
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...)```
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.

### 2.1.18 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 plaintext 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).

```julia
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.

```julia
displayable(mime)
displayable(d::Display, mime)
```

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.

```julia
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:

```julia
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"mime"` notation only supports literal strings; to construct MIME types in a more flexible manner use `MIME{symbol("mime")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function:

```julia
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.

```julia
minewritable(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)`.)

```julia
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`.

```julia
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 \( \text{Display} \). Then, for each MIME type (\( \text{mime} \) string) that can be displayed on \( D \), one should define a function \( \text{display}(d::D, ::\text{MIME"mime"}, x) = \ldots \) that displays \( x \) as that MIME type, usually by calling \( \text{reprmime}(\text{mime}, x) \). A \text{MethodError} should be thrown if \( x \) cannot be displayed as that MIME type; this is automatic if one calls \( \text{reprmime} \).

Finally, one should define a function \( \text{display}(d::D, x) \) that queries \( \text{mimewritable}(\text{mime}, x) \) for the \( \text{mime} \) types supported by \( D \) and displays the “best” one; a \text{MethodError} should be thrown if no supported MIME types are found for \( x \). Similarly, some subtypes may wish to override \( \text{redisplay}(d::D, \ldots) \). (Again, one should import \( \text{Base.display} \) to add new methods to \( \text{display} \).) 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 \( \text{display}(x) \) simply by pushing a new display onto the display-backend stack with:

\[
\text{pushdisplay}(d::\text{Display})
\]

Pushes a new display \( d \) on top of the global display-backend stack. Calling \( \text{display}(x) \) or \( \text{display}(\text{mime}, x) \) will display \( x \) on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a \text{MethodError}).

\[
\text{popdisplay}()
\]

\[
\text{popdisplay}(d::\text{Display})
\]

Pop the topmost backend off of the display-backend stack, or the topmost copy of \( d \) in the second variant.

\[
\text{TextDisplay}(\text{stream})
\]

Returns a \( \text{TextDisplay} <: \text{Display} \), 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.)

\[
\text{istext}(m::\text{MIME})
\]

Determine whether a MIME type is text data.

### 2.1.19 Memory-mapped I/O

\[
\text{mmap_array}(\text{type}, \text{dims}, \text{stream}[\ldots, \text{offset}])
\]

Create an \( \text{Array} \) 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).

\( \text{dims} \) 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.

**Example:**

```sh
# Create a file for mmapping
# (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

```julia
default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = mmap_array(Int, (m,n), s)
```

This would create 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).

```julia
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:**

```julia
B = mmap_bitarray((25,30000), s)
```

This would create a 25-by-30000 `BitArray`, linked to the file associated with stream s.

```julia
msync(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

```julia
msync(ptr[, len], flags)
```

Forces synchronization of the mmap’d 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.

```julia
MS_ASYNC
```

Enum constant for `msync`. See your platform man page for details. (not available on Windows).

```julia
MS_SYNC
```

Enum constant for `msync`. See your platform man page for details. (not available on Windows).

```julia
MS_INVALIDATE
```

Enum constant for `msync`. See your platform man page for details. (not available on Windows).

```julia
mmap(len, prot, flags, fd, offset)
```

Low-level interface to the mmap system call. See the man page.

```julia
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.

## 2.1.20 Standard Numeric Types

- `Bool` Int8 Uint8 Int16 Uint16 Int32 Uint32 Int64 Uint64 Int128 Uint128 Float16 Float32 Float64 Complex64 Complex128
2.1.21 Mathematical Operators

\[- (x)\]
Unary minus operator.

\[+ (x, y)\]
Binary addition operator.

\[- (x, y)\]
Binary subtraction operator.

\[\times (x, y)\]
Binary multiplication operator.

\[/ (x, y)\]
Binary left-division operator.

\[\backslash (x, y)\]
Binary right-division operator.

\[^ {(x, y)}\]
Binary exponentiation operator.

\[.+ (x, y)\]
Element-wise binary addition operator.

\[.- (x, y)\]
Element-wise binary subtraction operator.

\[.* (x, y)\]
Element-wise binary multiplication operator.

\[./ (x, y)\]
Element-wise binary left division operator.

\[\backslash. (x, y)\]
Element-wise binary right division operator.

\[.^{(x, y)}\]
Element-wise binary exponentiation operator.

\[\text{div} (a, b)\]
Compute \(a/b\), truncating to an integer

\[\text{fld} (a, b)\]
Largest integer less than or equal to \(a/b\)

\[\text{mod} (x, m)\]
Modulus after division, returning in the range \([0,m)\)

\[\text{rem} (x, m)\]
Remainder after division

\[\text{divrem} (x, y)\]
Compute \(x/y\) and \(x\%y\) at the same time

\[\% (x, m)\]
Remainder after division. The operator form of \text{rem}.

\[\text{modl} (x, m)\]
Modulus after division, returning in the range \((0,m]\)

\[\text{reml} (x, m)\]
Remainder after division, returning in the range \((0,m]\)
### rationalize([Type], x)
 Approximate the number x as a rational fraction

- `num(x)`: Numerator of the rational representation of x
- `den(x)`: Denominator of the rational representation of x

- `<< (x, n)`: Left shift operator.
- `>> (x, n)`: Right shift operator.
- `>>> (x, n)`: Unsigned right 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.

- `== (x, y)`: Numeric equality operator. Compares numbers and number-like values (e.g. arrays) by numeric value. True for numbers of different types that represent the same value (e.g. 2 and 2.0). Follows IEEE semantics for floating-point numbers. New numeric types should implement this function for two arguments of the new type.

- `!= (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)`: Equivalent to `!is(x, y)`

- `!== (x, y)`: See the `is()` operator

- `< (x, y)`: Less-than comparison operator. New numeric types should implement this function for two arguments of the new type.

- `<=(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)`: Greater-than-or-equals comparison operator.

- `.== (x, y)`: Element-wise equality comparison operator.

- `.!= (x, y)`: Element-wise not-equals comparison operator.
.\lt (x, y)
    Element-wise less-than comparison operator.

.\leq (x, y)
    Element-wise less-than-or-equals comparison operator.

.\gt (x, y)
    Element-wise greater-than comparison operator.

.\geq (x, y)
    Element-wise greater-than-or-equals comparison operator.

cmp (x, y)
    Return -1, 0, or 1 depending on whether x\lt y, x\equiv y, or x\gt y, respectively.

\sim (x)
    Bitwise not

& (x, y)
    Bitwise and

| (x, y)
    Bitwise or

\bitor (x, y)
    Bitwise exclusive or

! (x)
    Boolean not

&& (x, y)
    Boolean and

|| (x, y)
    Boolean or

A\ldiv_Bc (a, b)
    Matrix operator A \div B^H

A\ldiv_Bt (a, b)
    Matrix operator A \div B^T

A\mul_B (...)
    Matrix operator A B

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

A\ldiv_B (...)
    Matrix operator \ldiv_B (...)

A\ldiv_Bc (...)
    Matrix operator A^H \div B

A\ldiv_Bc (...)
    Matrix operator A^H \div B^H

2.1. Built-ins
\textbf{Ac\_mul\_B} (...)  
Matrix operator $A^H B$

\textbf{Ac\_mul\_Bc} (...)  
Matrix operator $A^H B^H$

\textbf{Ac\_rdiv\_B} ($a, b$)  
Matrix operator $A^H / B$

\textbf{Ac\_rdiv\_Bc} ($a, b$)  
Matrix operator $A^H / B^H$

\textbf{At\_ldiv\_B} (...)  
Matrix operator $A^T \backslash B$

\textbf{At\_ldiv\_Bt} (...)  
Matrix operator $A^T \backslash B^T$

\textbf{At\_mul\_B} (...)  
Matrix operator $A^T B$

\textbf{At\_mul\_Bt} (...)  
Matrix operator $A^T B^T$

\textbf{At\_rdiv\_B} ($a, b$)  
Matrix operator $A^T / B$

\textbf{At\_rdiv\_Bt} ($a, b$)  
Matrix operator $A^T / B^T$

\subsubsection*{2.1.22 Mathematical Functions}

\textbf{isapprox} ($x::\text{Number}, y::\text{Number}; \text{rtol}::\text{Real}=\text{cbrt}(\text{maxeps}), \text{atol}::\text{Real}=\text{sqrt}(\text{maxeps})$)  
Inexact equality comparison - behaves slightly different depending on types of input args:

- For \texttt{FloatingPoint} numbers, \texttt{isapprox} returns \texttt{true} if $\text{abs}(x-y) \leq \text{atol} + \text{rtol} \cdot \max(\text{abs}(x), \text{abs}(y))$.
- For \texttt{Integer} and \texttt{Rational} numbers, \texttt{isapprox} returns \texttt{true} if $\text{abs}(x-y) \leq \text{atol}$. The \texttt{rtol} argument is ignored. If one of $x$ and $y$ is \texttt{FloatingPoint}, the other is promoted, and the method above is called instead.
- For \texttt{Complex} numbers, the distance in the complex plane is compared, using the same criterion as above.

For default tolerance arguments, $\maxeps = \max(\text{eps}(\text{abs}(x)), \text{eps}(\text{abs}(y)))$.

\texttt{sin} ($x$)  
Compute sine of $x$, where $x$ is in radians

\texttt{cos} ($x$)  
Compute cosine of $x$, where $x$ is in radians

\texttt{tan} ($x$)  
Compute tangent of $x$, where $x$ is in radians

\texttt{sind} ($x$)  
Compute sine of $x$, where $x$ is in degrees

\texttt{cosd} ($x$)  
Compute cosine of $x$, where $x$ is in degrees

\texttt{tand} ($x$)  
Compute tangent of $x$, where $x$ is in degrees
\textbf{\texttt{sinpi} (x)}
Compute \(\sin(\pi x)\) more accurately than \(\sin(\pi^*x)\), especially for large \(x\).

\textbf{\texttt{cospi} (x)}
Compute \(\cos(\pi x)\) more accurately than \(\cos(\pi^*x)\), especially for large \(x\).

\textbf{\texttt{sinh} (x)}
Compute hyperbolic sine of \(x\)

\textbf{\texttt{cosh} (x)}
Compute hyperbolic cosine of \(x\)

\textbf{\texttt{tanh} (x)}
Compute hyperbolic tangent of \(x\)

\textbf{\texttt{asin} (x)}
Compute the inverse sine of \(x\), where the output is in radians

\textbf{\texttt{acos} (x)}
Compute the inverse cosine of \(x\), where the output is in radians

\textbf{\texttt{atan} (x)}
Compute the inverse tangent of \(x\), where the output is in radians

\textbf{\texttt{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.

\textbf{\texttt{asind} (x)}
Compute the inverse sine of \(x\), where the output is in degrees

\textbf{\texttt{acosd} (x)}
Compute the inverse cosine of \(x\), where the output is in degrees

\textbf{\texttt{atand} (x)}
Compute the inverse tangent of \(x\), where the output is in degrees

\textbf{\texttt{sec} (x)}
Compute the secant of \(x\), where \(x\) is in radians

\textbf{\texttt{csc} (x)}
Compute the cosecant of \(x\), where \(x\) is in radians

\textbf{\texttt{cot} (x)}
Compute the cotangent of \(x\), where \(x\) is in radians

\textbf{\texttt{secd} (x)}
Compute the secant of \(x\), where \(x\) is in degrees

\textbf{\texttt{csd} (x)}
Compute the cosecant of \(x\), where \(x\) is in degrees

\textbf{\texttt{cotd} (x)}
Compute the cotangent of \(x\), where \(x\) is in degrees

\textbf{\texttt{asec} (x)}
Compute the inverse secant of \(x\), where the output is in radians

\textbf{\texttt{acsc} (x)}
Compute the inverse cosecant of \(x\), where the output is in radians

\textbf{\texttt{acot} (x)}
Compute the inverse cotangent of \(x\), where the output is in radians
Compute the inverse secant of $x$, where the output is in degrees

Compute the inverse cosecant of $x$, where the output is in degrees

Compute the inverse cotangent of $x$, where the output is in degrees

Compute the hyperbolic secant of $x$

Compute the hyperbolic cosecant of $x$

Compute the hyperbolic cotangent of $x$

Compute the inverse hyperbolic sine of $x$

Compute the inverse hyperbolic cosine of $x$

Compute the inverse hyperbolic tangent of $x$

Compute the inverse hyperbolic secant of $x$

Compute the inverse hyperbolic cosecant of $x$

Compute the inverse hyperbolic cotangent of $x$

Compute \(\frac{\sin(\pi x)}{\pi x}\) if $x \neq 0$, and 1 if $x = 0$.

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($x$).

Convert $x$ from degrees to radians

Convert $x$ from radians to degrees

Compute the $\sqrt{x^2 + y^2}$ avoiding overflow and underflow

Compute the natural logarithm of $x$. Throws DomainError for negative Real arguments. Use complex negative arguments instead.

Compute the logarithm of $x$ to base 2. Throws DomainError for negative Real arguments.

Compute the logarithm of $x$ to base 10. Throws DomainError for negative Real arguments.

Accurate natural logarithm of $1+x$. Throws DomainError for Real arguments less than -1.
frexp(val, exp)
Return a number \(x\) such that it 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, 3, 2) 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.

clamp(x, lo, hi)
Return \(x\) if \(lo \leq x \leq hi\). If \(x < lo\), return \(lo\). If \(x > hi\), return \(hi\).
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 1 if the value of the sign of \( x \) is negative, otherwise 0.

flipsign \( (x, y) \)
    Return \( x \) with its sign flipped if \( y \) is negative. For example abs(x) = flipsign(x,x).

sqrt \( (x) \)
    Return \( \sqrt{x} \). Throws DomainError for negative Real arguments. Use complex negative arguments instead.

isqrt \( (x) \)
    Integer square root.

cbrt \( (x) \)
    Return \( x^{1/3} \)

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 \( \cos(z) + i \cdot \sin(z) \) if \( z \) is real. Return \( (\cos(\text{real}(z)) + i \cdot \sin(\text{real}(z))) / \text{exp}(\text{imag}(z)) \) if \( z \) is complex.

\( \text{binomial}(n,k) \)  
Number of ways to choose \( k \) out of \( n \) items.

\( \text{factorial}(n) \)  
Factorial of \( n \).

\( \text{factorial}(n,k) \)  
Compute \( \text{factorial}(n)/\text{factorial}(k) \).

\( \text{factor}(n) \)  
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.

**Example:** \( 100 = 2 \cdot 2 \cdot 5 \cdot 5; \) then, \( \text{factor}(100) \rightarrow [5=>2, 2=>2] \)

\( \text{gcd}(x,y) \)  
Greatest common (positive) divisor (or zero if \( x \) and \( y \) are both zero).

\( \text{lcm}(x,y) \)  
Least common (non-negative) multiple.

\( \text{gcdx}(x,y) \)  
Greatest common (positive) divisor, also returning integer coefficients \( u \) and \( v \) that solve \( ux + vy = \text{gcd}(x,y) \).

\( \text{ispow2}(n) \)  
Test whether \( n \) is a power of two.

\( \text{nextpow2}(n) \)  
Next power of two not less than \( n \).

\( \text{prevpow2}(n) \)  
Previous power of two not greater than \( n \).

\( \text{nextpow}(a,n) \)  
Next power of \( a \) not less than \( n \).

\( \text{prevpow}(a,n) \)  
Previous power of \( a \) not greater than \( n \).

\( \text{nextprod}([k_1,k_2,...],n) \)  
Next integer not less than \( n \) that can be written as \( \prod k_i^{p_i} \) for integers \( p_1, p_2, \text{etc.} \).

\( \text{prevprod}([k_1,k_2,...],n) \)  
Previous integer not greater than \( n \) that can be written as \( \prod k_i^{p_i} \) for integers \( p_1, p_2, \text{etc.} \).

\( \text{invmod}(x,m) \)  
Take the inverse of \( x \) modulo \( m \): \( y \) such that \( xy = 1 \pmod m \).

\( \text{powermod}(x,p,m) \)  
Compute \( x^p \pmod m \).

\( \text{gamma}(x) \)  
Compute the gamma function of \( x \).

\( \text{lgamma}(x) \)  
Compute the logarithm of absolute value of \( \text{gamma}(x) \).
lfact (x)
    Compute the logarithmic factorial of \( x \)

digamma (x)
    Compute the digamma function of \( x \) (the logarithmic derivative of \( \text{gamma}(x) \) )

invdigamma (x)
    Compute the inverse digamma function of \( x \).

trigamma (x)
    Compute the trigamma function of \( x \) (the logarithmic second derivative of \( \text{gamma}(x) \) )

polygamma \((m, x)\)
    Compute the polygamma function of order \( m \) of argument \( x \) (the \((m+1)\)th derivative of the logarithm of \( \text{gamma}(x) \) )

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 \( \text{Bi}'(x) \).

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 \((\nu, x)\)
    Bessel function of the first kind of order \( \nu \), \( J_\nu(x) \).

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 \((\nu, x)\)
    Bessel function of the second kind of order \( \nu \), \( Y_\nu(x) \).

hankelh1 \((\nu, x)\)
    Bessel function of the third kind of order \( \nu \), \( H^{(1)}_\nu(x) \).

hankelh2 \((\nu, x)\)
    Bessel function of the third kind of order \( \nu \), \( H^{(2)}_\nu(x) \).

besseln \((\nu, k, x)\)
    Bessel function of the third kind of order \( \nu \) (Hankel function). \( k \) is either 1 or 2, selecting \( \text{hankelh1} \) or \( \text{hankelh2} \), respectively.

besseli \((\nu, x)\)
    Modified Bessel function of the first kind of order \( \nu \), \( I_\nu(x) \).
**besselk** \((\nu, x)\)
Modified Bessel function of the second kind of order \(\nu\), \(K_\nu(x)\).

**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** \((x)\)
Riemann zeta function \(\zeta(s)\).

**bitmix** \((x, y)\)
Hash two integers into a single integer. Useful for constructing hash functions.

**ndigits** \((n, b)\)
Compute the number of digits in number \(n\) written in base \(b\).

### 2.1.23 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
\textbf{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.

\textbf{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.

\textbf{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.

\textbf{signed} (x)

Convert a number to a signed integer

\textbf{unsigned} (x)

Convert a number to an unsigned integer

\textbf{int8} (x)

Convert a number or array to \texttt{Int8} data type

\textbf{int16} (x)

Convert a number or array to \texttt{Int16} data type

\textbf{int32} (x)

Convert a number or array to \texttt{Int32} data type

\textbf{int64} (x)

Convert a number or array to \texttt{Int64} data type

\textbf{int128} (x)

Convert a number or array to \texttt{Int128} data type

\textbf{uint8} (x)

Convert a number or array to \texttt{Uint8} data type

\textbf{uint16} (x)

Convert a number or array to \texttt{Uint16} data type

\textbf{uint32} (x)

Convert a number or array to \texttt{Uint32} data type

\textbf{uint64} (x)

Convert a number or array to \texttt{Uint64} data type

\textbf{uint128} (x)

Convert a number or array to \texttt{Uint128} data type

\textbf{float16} (x)

Convert a number or array to \texttt{Float16} data type

\textbf{float32} (x)

Convert a number or array to \texttt{Float32} data type

\textbf{float64} (x)

Convert a number or array to \texttt{Float64} data type

\textbf{float32\_isvalid} \((x, \text{out::Vector\{Float32\}}) \rightarrow \text{Bool}\)

Convert a number or array to \texttt{Float32} data type, returning true if successful. The result of the conversion is stored in \texttt{out[1]}.
```haskell
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. For strings, it converts to Float64.

significand(x)
  Extract the significand(s) (a.k.a. mantissa), in binary representation, of a floating-point number or array.
  For example, significand(15.2)/15.2 == 0.125, and 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

complex128(r, i)
  Convert to r+i*im represented as a Complex128 data type

char(x)
  Convert a number or array to Char data type

complex(r, i)
  Convert real numbers or arrays to complex

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 ASCII-IAString.

2.1.24 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
  The constant pi

im
  The imaginary unit

e
  The constant e
```

### 2.1. Built-ins
catalan
    Catalan’s constant

Inf
    Positive 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) → Bool
    Test whether a floating point number is subnormal

isfinite (f) → Bool
    Test whether a number is finite

isinf (f)
    Test whether a number is infinite

isnan (f)
    Test whether a floating point number is not a number (NaN)

inf (f)
    Returns infinity in the same floating point type as f (or f can by the type itself)

nan (f)
    Returns NaN in the same floating point type as f (or f can by the type itself)

nextfloat (f)
    Get the next floating point number in lexicographic order

prevfloat (f) → Float
    Get the previous floating point number in lexicographic order

isinteger (x)
    Test whether x or all its elements are numerically equal to some integer

isreal (x)
    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()
Get the current floating point rounding mode. Valid modes are RoundNearest, RoundToZero, RoundUp and RoundDown.

set_rounding(mode)
Set the floating point rounding mode. See get_rounding for available modes

with_rounding(f::Function, mode)
Change the floating point rounding mode for the duration of f. It is logically equivalent to:

```julia
old = get_rounding()
set_rounding(mode)
f()
set_rounding(old)
```

See get_rounding for available rounding modes.

Integers

count_ones(x::Integer) → Integer
Number of ones in the binary representation of x.
Example: count_ones(7) -> 3

count_zeros(x::Integer) → Integer
Number of zeros in the binary representation of x.
Example: count_zeros(int32(2 ^ 16 - 1)) -> 16

leading_zeros(x::Integer) → Integer
Number of zeros leading the binary representation of x.
Example: leading_zeros(int32(1)) -> 31

leading_ones(x::Integer) → Integer
Number of ones leading the binary representation of x.
Example: leading_ones(int32(2 ^ 32 - 2)) -> 31

trailing_zeros(x::Integer) → Integer
Number of zeros trailing the binary representation of x.
Example: trailing_zeros(2) -> 1

trailing_ones(x::Integer) → Integer
Number of ones trailing the binary representation of x.
Example: trailing_ones(3) -> 2

isprime(x::Integer) → Bool
Returns true if x is prime, and false otherwise.
Example: isprime(3) -> true

primes(n)
Returns a collection of the prime numbers <= n.

isodd(x::Integer) → Bool
Returns true if x is odd (that is, not divisible by 2), and false otherwise.
Example: isodd(9) -> false
**Julia Language Documentation, Release 0.2.0**

**iseven** \((x::Integer) \rightarrow \text{Bool}\)

- Returns `true` if \(x\) is even (that is, divisible by 2), and `false` otherwise.
- **Example:** `iseven(1) -> false`

### 2.1.25 BigFloats

The **BigFloat** type implements arbitrary-precision floating-point arithmetic using the [GNU MPFR library](https://fredrik.allert.org/).

**precision** \((\text{num::FloatingPoint})\)

- 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** \((x::\text{Int64})\)

- Set the precision (in bits) to be used to BigFloat arithmetic.

**with_bigfloat_precision** \((f::\text{Function}, \text{precision::Integer})\)

- Change the BigFloat arithmetic precision (in bits) for the duration of \(f\). It is logically equivalent to:

  
  \[
  \begin{align*}
  \text{old} &= \text{get_bigfloat_precision}() \\
  \text{set_bigfloat_precision}(\text{precision}) \\
  f() \\
  \text{set_bigfloat_precision}(\text{old})
  \end{align*}
  \]

**get_bigfloat_rounding**()

- Get the current BigFloat rounding mode. Valid modes are `RoundNearest`, `RoundToZero`, `RoundUp`, `RoundDown`, `RoundFromZero`.

**set_bigfloat_rounding** \((\text{mode})\)

- Set the BigFloat rounding mode. See `get_bigfloat_rounding` for available modes.

**with_bigfloat_rounding** \((f::\text{Function}, \text{mode})\)

- Change the BigFloat rounding mode for the duration of \(f\). See `get_bigfloat_rounding` for available rounding modes; see also `with_bigfloat_precision`.

### 2.1.26 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.

**srand** \([rng, seed]\)

- 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**()

- 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 srand.

**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 from the inclusive interval specified by Rangel r (for example, 1:n). 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** *(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!** *(A::Array{Float64,N})*
Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the rand function.

**randsym** *(n)*
Generate an nxn symmetric array of normally-distributed random numbers with mean 0 and standard deviation 1.

### 2.1.27 Arrays

#### 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

**nnz** *(A)*
Counts the number of nonzero values in array A (dense or sparse)

**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.

\textit{ind2sub} \((\text{dims, index}) \rightarrow \text{subscripts}\)

Returns a tuple of subscripts into an array with dimensions \(\text{dims}\), corresponding to the linear index \(\text{index}\).

\textbf{Example}

\(i, j, \ldots = \text{ind2sub(size(A), indmax(A))}\) provides the indices of the maximum element.

\textit{sub2ind} \((\text{dims, i, j, \ldots}) \rightarrow \text{index}\)

The inverse of \textit{ind2sub}, returns the linear index corresponding to the provided subscripts.

\textbf{Constructors}

\textit{Array} \((\text{type, dims})\)

Construct an uninitialized dense array. \(\text{dims}\) may be a tuple or a series of integer arguments.

\textit{getindex} \((\text{type[\ldots, elements\ldots]}\))

Construct a 1-d array of the specified type. This is usually called with the syntax \(\text{Type[]}\). Element values can be specified using \(\text{Type}[a, b, c, \ldots]\).

\textit{cell} \((\text{dims})\)

Construct an uninitialized cell array (heterogeneous array). \(\text{dims}\) can be either a tuple or a series of integer arguments.

\textit{zeros} \((\text{type, dims})\)

Create an array of all zeros of specified type.

\textit{ones} \((\text{type, dims})\)

Create an array of all ones of specified type.

\textit{infs} \((\text{type, dims})\)

Create an array where every element is infinite and of the specified type.

\textit{nans} \((\text{type, dims})\)

Create an array where every element is NaN of the specified type.

\textit{trues} \((\text{dims})\)

Create a \textit{BitArray} with all values set to true.

\textit{false} \((\text{dims})\)

Create a \textit{BitArray} with all values set to false.

\textit{fill} \((v, \text{dims})\)

Create an array filled with \(v\).

\textit{fill!} \((A, x)\)

Fill array \(A\) with value \(x\).

\textit{reshape} \((A, \text{dims})\)

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.

\textit{similar} \((\text{array, element_type, dims})\)

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 \(\text{dims}\) argument may be a tuple or a series of integer arguments.

\textit{reinterpret} \((\text{type, A})\)

Change the type-interpretation of a block of memory. For example, \textit{reinterpret}(Float32,
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

**linspace** \((\text{start}, \text{stop}, n)\)
- Construct a vector of \(n\) linearly-spaced elements from \(\text{start}\) to \(\text{stop}\).

**logspace** \((\text{start}, \text{stop}, n)\)
- Construct a vector of \(n\) logarithmically-spaced numbers from \(10^{\text{start}}\) to \(10^{\text{stop}}\).

### Mathematical operators and functions

All mathematical operations and functions are supported for arrays.

**broadcast** \((f, A, ...)_s\)
- Broadcasts the arrays \(A\) to a common size by expanding singleton dimensions, and returns an array of the results \(f(a...\)) for each position.

**broadcast!** \((f, \text{dest}, A, ...)_s\)
- Like `broadcast`, but store the result of `broadcast(f, A, ...)` 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)`.

**broadcast_function** \((f)\)
- Returns a function `broadcast_f` such that `broadcast_function(f)(A, ...)` === `broadcast(f, A, ...)`. Most useful in the form `const broadcast_f = broadcast_function(f)`.

**broadcast!_function** \((f)\)
- Like `broadcast_function`, but for `broadcast!`.

### Indexing, Assignment, and Concatenation

**getindex** \((A, \text{inds}\, ...)_s\)
- Returns a subset of array \(A\) as specified by \(\text{inds}\), where each \(\text{ind}\) may be an \(\text{Int}\), a \(\text{Range}\), or a \(\text{Vector}\).

**sub** \((A, \text{inds}\, ...)_s\)
- Returns a SubArray, which stores the input \(A\) and \(\text{inds}\) 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** \((A, \text{inds}\, ...)_s\)
- Create a view of the given indexes of array \(A\), dropping dimensions indexed with scalars.

**setindex!** \((A, X, \text{inds}\, ...)_s\)
- Store values from array \(X\) within some subset of \(A\) as specified by \(\text{inds}\).
broadcast_getindex \((A, \text{inds}...)\)
Broadcasts the \text{inds} arrays to a common size like \text{broadcast}, and returns an array of the results \(A[\text{ks}...]\), where \text{ks} goes over the positions in the broadcast.

broadcast_setindex! \((A, X, \text{inds}...)\)
Broadcasts the \(X\) and \text{inds} arrays to a common size and stores the value from each position in \(X\) at the indices given by the same positions in \text{inds}.

cat \((\text{dim}, A...)\)
Concatenate the input arrays along the specified dimension

vcat \((A...)\)
Concatenate along dimension 1

hcat \((A...)\)
Concatenate along dimension 2

hvcat \((\text{rows}::(\text{Int}...), \text{values}...)\)
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 \text{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 \text{flipdim}(A,1).

fliplr \((A)\)
Equivalent to \text{flipdim}(A,2).

circshift \((A, \text{shifts})\)
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 \(A\).

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 \(A\).

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.

nonzeros \((A)\)
Return a vector of the non-zero values in array \(A\).

findfirst \((A)\)
Return the index of the first non-zero value in \(A\).

findfirst \((A, v)\)
Return the index of the first element equal to \(v\) in \(A\).

findfirst \((\text{predicate}, A)\)
Return the index of the first element that satisfies the given predicate in \(A\).
`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 $A$ satisfying the given predicate, 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 $\text{ndims}(A)$. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to $\text{permute}(A, [2, 1])$.

`ipermutedims(A, perm)`  
Like `permutedims()`, except the inverse of the given permutation is applied.

`squeeze(A, dims)`  
Remove the dimensions specified by $\text{dims}$ from array $A$

`vec(Array) → 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.

### Array functions

`cumprod(A[, dim])`  
Cumulative product along a dimension.

`cumsum(A[, dim])`  
Cumulative sum along a dimension.

`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.

`rot90(A)`  
Rotate matrix $A$ left 90 degrees.

`rotr90(A)`  
Rotate matrix $A$ right 90 degrees.
**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.

**mapslices** (f, A, dims)

Transform the given dimensions of array A using function f. f is called on each slice of A of the form A[...,:,...,:,...]. dims is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if dims is [1,2] and A is 4-dimensional, f is called on A[;,:,i,j] for all i and j.

**sum_kbn** (A)

Returns the sum of all array elements, using the Kahan-Babushka-Neumaier compensated summation algorithm for additional accuracy.

**cartesianmap** (f, dims)

Given a dims tuple of integers (m, n, ...), call f on all combinations of integers in the ranges 1:m, 1:n, etc. Example:

```
julia> cartesianmap println, (2,2)
11
21
12
22
```

**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.

### 2.1.28 Combinatorics

**nthperm** (v, k)

Compute the kth lexicographic permutation of a vector.

**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!** 
\( v, p \)

Permutes 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** 
\( n \)

Constructs a random cyclic permutation of the given length.

**shuffle** 
\( v \)

Returns a randomly permuted copy of \( v \).

**shuffle!** 
\( v \)

In-place version of **shuffle()**.

**reverse** 
\( v \), \( start=1 \), \( stop=length(v) \)

Returns a copy of \( v \) reversed from \( start \) to \( stop \).

**reverse!** 
\( v \), \( start=1 \), \( stop=length(v) \) → \( v \)

In-place version of **reverse()**.

**combinations** 
\( iir, n \)

Generate all combinations of \( n \) elements from a given iterable 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** 
\( iir \)

Generate all permutations of a given iterable 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))`.

### 2.1.29 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.
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 \( \sqrt{\frac{\sum((v - \text{mean}(v))^2)}{\text{length}(v) - 1}} \). 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 \( \frac{\sum((v - \text{mean}(v))^2)}{\text{length}(v) - 1} \). 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.

median(v; checknan::Bool=true)
Compute the median of a vector v. If keyword argument checknan is true (the default), an error is raised for data containing NaN values. Note: Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the DataArray package is recommended.

median!(v; checknan::Bool=true)
Like median, but may overwrite the input vector.

hist(v[, n]) \rightarrow 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) \rightarrow 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.

hist2d(M, e1, e2) \rightarrow (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.

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** \((v1[\cdot, v2[\cdot)\)
Compute the Pearson covariance between two vectors \(v1\) and \(v2\). If called with a single element \(v\), then computes covariance of columns of \(v\). Note: Julia does not ignore NaN values in the computation.

**cor** \((v1[\cdot, v2[\cdot)\)
Compute the Pearson correlation between two vectors \(v1\) and \(v2\). If called with a single element \(v\), then computes correlation of columns of \(v\). Note: Julia does not ignore NaN values in the computation.

### 2.1.30 Signal Processing

FFT functions in Julia are largely implemented by calling functions from FFTW

**fft** \((A[\cdot, \text{dims}[\cdot)\)
Performs a multidimensional FFT of the array \(A\). The optional \text{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 \text{nextprod}(). See also \text{plan_fft}() for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

\[
\text{DFT}[k] = \sum_{n=1}^{\text{length}(A)} \exp \left( -i \frac{2\pi(n-1)(k-1)}{\text{length}(A)} \right) A[n].
\]

A multidimensional FFT simply performs this operation along each transformed dimension of \(A\).

**fft!** \((A[\cdot, \text{dims}[\cdot)\)
Same as **fft()**, but operates in-place on \(A\), which must be an array of complex floating-point numbers.

**ifft** \((A[\cdot, \text{dims}[\cdot)\)
Multidimensional inverse FFT.

A one-dimensional backward FFT computes \(\text{BDFT}[k] = \sum_{n=1}^{\text{length}(A)} \exp \left( +i \frac{2\pi(n-1)(k-1)}{\text{length}(A)} \right) A[n].\) A multidimensional backward FFT simply performs this operation along each transformed dimension of \(A\). The inverse FFT computes the same thing divided by the product of the transformed dimensions.

**ifft!** \((A[\cdot, \text{dims}[\cdot)\)
Same as **ifft()**, but operates in-place on \(A\).

**bfft** \((A[\cdot, \text{dims}[\cdot)\)
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.)

**bfft!** \((A[\cdot, \text{dims}[\cdot)\)
Same as **bfft()**, but operates in-place on \(A\).

**plan_fft** \((A[\cdot, \text{dims}[\cdot, \text{flags}[\cdot, \text{timelimit}[\cdot)\]
Pre-plan an optimized FFT along given dimensions \(\text{dims}\) of arrays matching the shape and type of \(A\). (The first two arguments have the same meaning as for **fft()**.) Returns a function \text{plan}(A)\) that computes \text{fft}(A, \text{dims})\) quickly.

The \text{flags} argument is a bitwise-or of FFTW planner flags, defaulting to FFTW.\text{ESTIMATE}. e.g. passing FFTW.\text{MEASURE} or FFTW.\text{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_ff!(A, dims[, flags[, timelimit]]])`
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_ff!(A)` but produces a plan that performs inverse transforms `ifft()`.

`plan_bfft (A[, dims[, flags[, timelimit]]])`
Same as `plan_ff!(A)` but produces a plan that performs an unnormalized backwards transform `bfft()`.

`plan_ff!(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 dimensions 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 \(d = \text{floor}(\text{size}(A,\text{dims}[1])/2)+1\). (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`.

\[
 filt (b, a, x)
\]
Apply filter described by vectors \( a \) and \( b \) to vector \( x \).

\[
deconv (b, a)
\]
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.REDFT10, or FFTW.REDFT11), a discrete sine transform of various types (FFTW.RODFT00, 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 > length(kind)`.

See also `plan_r2r()` to pre-plan optimized `r2r` transforms.

```text
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.
```

```text
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.
```

```text
plan_r2r!(A, kind[\], dims[], flags[], timelimit[])  
Similar to `Base.plan_fft()`, but corresponds to `r2r!()`.
```

## 2.1.31 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.

```text
quadgk(f, a, b, c...; reltol=sqrt(eps), abstol=0, maxevals=10^7, order=7)  
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 `reltol` (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 either E <= abstol or E <= reltol*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).

The algorithm is an adaptive Gauss-Kronrod integration technique: the integral in each interval is estimated using a Kronrod rule (2*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.)

### 2.1.32 Parallel Computing

```julia
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.
```

```julia
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]. A worker is started for each such definition.

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 -i /home/foo/bar.pem'.
```

```julia
nprocs()  
Get the number of available processors.
```

```julia
nworkers()  
Get the number of available worker processors. This is one less than nprocs(). Equal to nproc() if nprocs() == 1.
```

```julia
procs()  
Returns a list of all process identifiers.
```

```julia
workers()  
Returns a list of all worker process identifiers.
```

```julia
rmprocs(pids...)  
Removes the specified workers.
```

```julia
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.
```

```julia
myid()  
Get the id of the current processor.
```

```julia
pmap(f, lsts...; err_retry=true, err_stop=false)  
Transform collections lsts 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.
```

```julia
remotecall(id, func, args...)  
Call a function asynchronously on the given arguments on the specified processor. 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)

**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`.

**take***(RemoteRef)**
Fetch the value of a remote reference, removing it so that the reference is empty again.

**isready***(RemoteRef)**
Determine whether a `RemoteRef` has a value stored to it. Note that this function can easily 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.

**RemoteRef()**
Make an uninitialized remote reference on the local machine.

**RemoteRef(n)**
Make an uninitialized remote reference on processor 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 processor, returning a `RemoteRef` to the result.

**@spawnat()**
Accepts two arguments, p and an expression, and runs the expression asynchronously on processor 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, and @spawnat` complete.
### 2.1.33 Distributed Arrays

**DArray**\((init, dims[, procs, dist])\)

Construct a distributed array. \(init\) 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 processor 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:

\[
dfill(v, \text{args}...) = \text{DArray}(I\rightarrow\text{fill}(v, \text{map}(\text{length},I)), \text{args}...)
\]

**dzeros**\((\text{dims}, ...)\)

Construct a distributed array of zeros. Trailing arguments are the same as those accepted by \(\text{darray}\).

**dones**\((\text{dims}, ...)\)

Construct a distributed array of ones. Trailing arguments are the same as those accepted by \(\text{darray}\).

**dfill**\((x, \text{dims}, ...)\)

Construct a distributed array filled with value \(x\). Trailing arguments are the same as those accepted by \(\text{darray}\).

**drand**\((\text{dims}, ...)\)

Construct a distributed uniform random array. Trailing arguments are the same as those accepted by \(\text{darray}\).

**drandn**\((\text{dims}, ...)\)

Construct a distributed normal random array. Trailing arguments are the same as those accepted by \(\text{darray}\).

**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.

**myindexes**\((d)\)

A tuple describing the indexes owned by the local processor. Returns a tuple with empty ranges if no local part exists on the calling process.

**procs**\((d)\)

Get the vector of processors storing pieces of \(d\)

### 2.1.34 System

**run**\((\text{command})\)

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

**spawn**\((\text{command})\)

Run a command object asynchronously, returning the resulting \text{Process} object.

**DevNull**

Used in a stream redirect to discard all data written to it. Essentially equivalent to \(/\text{dev/null}\) on Unix or NUL on Windows. Usage: \(\text{run('cat test.txt' |> DevNull)}\)

**success**\((\text{command})\)

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, signum=SIGTERM)*
Send a signal to a process. The default is to terminate the process.

**readfrom**(command)
Starts running a command asynchronously, and returns a tuple (stream,process). The first value is a stream reading from the process’ standard output.

**writeto**(command)
Starts running a command asynchronously, and returns a tuple (stream,process). The first value is a stream writing to the process’ standard input.

**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)
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".

|> *(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')

```python
|> *(command, filename)*
Redirect standard output of a process, appending to the destination file.
```

```python
.|*(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”.

**pwd** () → String
Get the current working directory.

**cd**(dir::String)
Set the current working directory. Returns the new current 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.

rmdir(path)
Remove the directory named path.

getpid() → Int32
Get julia’s process ID.

time([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 and expression, printing 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() \to 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.

2.1.35 C Interface

call((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:

```julia
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_LOCAL 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.

**c_malloc** *(size::Integer)*
Call **malloc** from the C standard library.

**c_free** *(addr::Ptr)*
Call **free** from the C standard library.

**unsafe_load** *(p::Ptr{T}, i::Integer)*
Derference the pointer p[i] or *p, returning a copy of type T.

**unsafe_store!** *(p::Ptr{T}, x, i::Integer)*
Assign to the pointer p[i] = x or *p = x, making a copy of object x into the memory at p.

**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.

**copy!** *(dest, src)*
Copy all elements from collection src to array dest.

**copy!** *(dest, do, src, so, N)*
Copy N elements from collection src starting at offset so, to array dest starting at offset do.

**pointer** *(a[, index ])*
Get the native address of an array 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:
```julia
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`.

**find_library** (*names, locations*)
Searches for the first library in `names` in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be `dlopen`'d. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a `global const` and used as the library name in future `ccall`'s. On failure, it returns the empty string.

**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.

**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)

2.1.36 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.

errno()
   Get the value of the C library’s errno

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

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.

2.1. Built-ins
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.

2.1.37 Tasks

Task (func)
Create a Task (i.e. thread, or coroutine) to execute the given function. 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 args. On subsequent switches, args are returned from the task’s last call to yieldto.

current_task ()
Get the currently running Task.

istaskdone (task)
Tell whether a task has exited.

consume (task)
Receive the next value passed to produce by the specified task.

produce (value)
Send the given value to the last consume call, switching to the consumer task.

yield ()
For scheduled tasks, switch back 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.

\[\text{task_local_storage}(body, symbol, value)\]
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\((condition, val=\text{nothing}; all=\text{true}, error=\text{false})\)
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\((t:\text{Task})\)
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.

@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.

### 2.1.38 Events

**Timer\((f:\text{Function})\)**
Create a timer to call the given callback function. The callback is passed two arguments: the timer object itself, and a status code, which will be 0 unless an error occurs. The timer can be started and stopped with start_timer and stop_timer.

\[\text{start_timer}(t:\text{Timer}, delay, repeat)\]
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.

\[\text{stop_timer}(t:\text{Timer})\]
Stop invoking the callback for a timer.

### 2.1.39 Reflection

module_name\((m:\text{Module}) \to \text{Symbol}\)
Get the name of a module as a symbol.

module_parent\((m:\text{Module}) \to \text{Module}\)
Get a module’s enclosing module. Main is its own parent.

current_module\()\to \text{Module}\)
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.

---

**2.1. Built-ins**

227
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 \((x)\)
Get an array of the names exported by a module, or the fields of a data type. 

isconst \([m::Module], s::Symbol \rightarrow \text{Bool}\)
Determine whether a global is declared `const` in a given module. 

isgeneric \((f::Function) \rightarrow \text{Bool}\)
Determine whether a function is generic. 

function_name \((f::Function) \rightarrow \text{Symbol}\)
Get the name of a generic function as a symbol, or `:anonymous`. 

function_module \((f::Function, \text{types}) \rightarrow \text{Module}\)
Determine the module containing a given definition of a generic function. 

functionloc \((f::Function, \text{types})\)
Returns a tuple \((\text{filename}, \text{line})\) giving the location of a method definition. 

functionlocs \((f::Function, \text{types})\)
Returns an array of the results of `functionloc` for all matching definitions. 

### 2.1.40 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** \((f, \text{types})\)
Returns an array of lowered ASTs for the methods matching the given generic function and type signature. 

**code_typeded** \((f, \text{types})\)
Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature. 

**code LLVM** \((f, \text{types})\)
Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to STDOUT. 

**code native** \((f, \text{types})\)
Prints the native assembly instructions generated for running the method matching the given generic function and type signature to STDOUT. 

**precompile** \((f, \text{args}::(\text{Any,...}))\)
Compile the given function \(f\) for the argument tuple (of types) \(\text{args}\), but do not execute it.
2.1.41 Sparse Matrices

Sparse matrices support much of the same set of operations as dense matrices. The following functions are specific to sparse matrices.

\texttt{\texttt{sparse}(I, J, V[, m, n, combine])}

Create a sparse matrix \( S \) of dimensions \( m \times n \) such that \( S[I[k], J[k]] = V[k] \). The \texttt{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 \texttt{combine} function is not supplied, duplicates are added by default.

\texttt{\texttt{sparsevec}(I, V[, m, combine])}

Create a sparse matrix \( S \) of size \( m \times 1 \) such that \( S[I[k]] = V[k] \). Duplicates are combined using the \texttt{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.

\texttt{\texttt{sparsevec}(D::Dict[, m])}

Create a sparse matrix of size \( m \times 1 \) where the row values are keys from the dictionary, and the nonzero values are the values from the dictionary.

\texttt{\texttt{issparse}(S)}

Returns \texttt{true} if \( S \) is sparse, and \texttt{false} otherwise.

\texttt{\texttt{sparse}(A)}

Convert a dense matrix \( A \) into a sparse matrix.

\texttt{\texttt{sparsevec}(A)}

Convert a dense vector \( A \) into a sparse matrix of size \( m \times 1 \). In julia, sparse vectors are really just sparse matrices with one column.

\texttt{\texttt{dense}(S)}

Convert a sparse matrix \( S \) into a dense matrix.

\texttt{\texttt{full}(S)}

Convert a sparse matrix \( S \) into a dense matrix.

\texttt{\texttt{spzeros}(m, n)}

Create an empty sparse matrix of size \( m \times n \).

\texttt{\texttt{spones}(S)}

Create a sparse matrix with the same structure as that of \( S \), but with every nonzero element having the value \( 1.0 \).

\texttt{\texttt{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 \( m \times n \).

\texttt{\texttt{spdiagm}(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.

\texttt{\texttt{sprand}(m, n, density[, rng])}

Create a random sparse matrix with the specified density. Nonzeros are sampled from the distribution specified by \( \texttt{rng} \). The uniform distribution is used in case \( \texttt{rng} \) is not specified.

\texttt{\texttt{sprandn}(m, n, density)}

Create a random sparse matrix of specified density with nonzeros sampled from the normal distribution.
sprandbool \((m, n, \text{density})\)
Create a random sparse boolean matrix with the specified density.

etree \((A[, , \text{post}])\)
Compute the elimination tree of a symmetric sparse matrix \(A\) from \(\text{triu}(A)\) 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.

### 2.1.42 Linear Algebra

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

\star \((A, B)\)
Matrix multiplication

\((A, B)\)
Matrix division using a polyalgorithm. For input matrices \(A\) and \(B\), the result \(X\) is such that \(AX = B\) 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 reducing \(A\) to bidiagonal form and solving the bidiagonal least squares problem. For sparse, square \(A\) the LU factorization (from UMFPACK) is used.

dot \((x, y)\)
Compute the dot product

cross \((x, y)\)
Compute the cross product of two 3-vectors

norm \((a)\)
Compute the norm of a Vector or a Matrix

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: \(A=\text{factorize}(A)\); \(x=A\backslash b\); \(y=A\backslash C\).

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 \(A[p, :] = L\cdot U\).

lufact \((A)\) → \(LU\)
Compute the LU factorization of \(A\), returning an \(LU\) object for dense \(A\) or an 
UmfpackLU object for sparse \(A\). The individual components of the factorization \(F\) can be accessed by indexing: \(F[:L], F[:U]\), and \(F[:P]\) (permutation matrix) or \(F[:p]\) (permutation vector). An UmfpackLU object has additional components \(F[:q]\) (the left permutation vector) and \(Rs\) the vector of scaling factors. The following functions are available for both LU and UmfpackLU objects: \(\text{size}, \\backslash \text{, and det. For LU there is also an \text{inv} method. The sparse LU factorization is such that } L\cdot U \text{ is equal to } \text{scale(Rs, A)}[p, q]^{\backslash }\).
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.

c chol(A[:,LU]) → F

Compute Cholesky factorization of a symmetric positive-definite matrix A and return the matrix F. If LU is L (Lower), A = L*L'. If LU is U (Upper), A = R' * R.

c cholfact!(A[:,LU]) → Cholesky

cholfact! is the same as cholfact(), but saves space by overwriting the input A, instead of creating a copy.

c cholpfact!(A[:,LU]) → CholeskyPivoted

cholpfact! is the same as cholpfact(), but saves space by overwriting the input A, instead of creating a copy.

c qr(A[:,thin]) → Q, R

Compute the QR factorization of A such that A = Q*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)

Computes the QR factorization of A and returns a QR type, which is a Factorization F consisting of an orthogonal matrix F[:Q] and a triangular matrix F[:R]. The following functions are available for QR objects: size, \. The orthogonal matrix Q=F[:Q] is a QRPackedQ type which has the * operator overloaded to support efficient multiplication by Q and Q’. 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 QRPackedQ matrix can be converted into a regular matrix with full.

qrfact!(A)

qrfact! is the same as qrfact(), but saves space by overwriting the input A, instead of creating a copy.

qrp(A[:,thin]) → Q, R, p

Computes the QR factorization of A with pivoting, such that A[:,p] = Q*R. Also see qrpfact. The default is to compute a thin factorization.
qrpfact \((A) \rightarrow \text{QRPivoted}\)
Computes the QR factorization of \(A\) with pivoting and returns a \text{QRPivoted} object, which is a Factorization \(F\) consisting of an orthogonal matrix \(F[:Q]\), a triangular matrix \(F[:R]\), and a permutation \(F[:p]\) (or its matrix representation \(F[:P]\)). The following functions are available for \text{QRPivoted} objects: size, \. The orthogonal matrix \(Q=F[:Q]\) is a \text{QRPivotedQ} type which has the + operator overloaded to support efficient multiplication by \(Q\) and \(Q'\). Multiplication with respect to either the thin or full \(Q\) is allowed, i.e. both \(F[:Q]*F[:R]\) and \(F[:Q]*A\) are supported. A \text{QRPivotedQ} matrix can be converted into a regular matrix with full.

qrpfact! \((A) \rightarrow \text{QRPivoted}\)
qrpfact! is the same as qrpfact(), but saves space by overwriting the input \(A\), instead of creating a copy.

bkfact \((A) \rightarrow \text{BunchKaufman}\)
Compute the Bunch Kaufman factorization of a real symmetric or complex Hermitian matrix \(A\) and return a \text{BunchKaufman} object. The following functions are available for \text{BunchKaufman} objects: size, \, inv, issym, ishermitian.

bkfact! \((A) \rightarrow \text{BunchKaufman}\)
bkfact! is the same as bkfact(), but saves space by overwriting the input \(A\), instead of creating a copy.

sqrtm \((A)\)
Compute the matrix square root of \(A\). If \(B = \text{sqrtm}(A)\), then \(B*B \equiv A\) within roundoff error.

eig \((A) \rightarrow D, V\)
Compute eigenvalues and eigenvectors of \(A\)

eig \((A, B) \rightarrow D, V\)
Compute generalized eigenvalues and vectors of \(A\) and \(B\)

eigvals \((A)\)
Returns the eigenvalues of \(A\).

eigmax \((A)\)
Returns the largest eigenvalue of \(A\).

eigmin \((A)\)
Returns the smallest eigenvalue of \(A\).

eigvecs \((A[, \text{eigvals}])\)
Returns the eigenvectors of \(A\).

For SymTridiagonal matrices, if the optional vector of eigenvalues eigvals is specified, returns the specific corresponding eigenvectors.

eigfact \((A)\)
Compute the eigenvalue decomposition of \(A\) and return an \text{Eigen} object. If \(F\) is the factorization object, the eigenvalues can be accessed with \(F[:values]\) and the eigenvectors with \(F[:vectors]\). The following functions are available for \text{Eigen} objects: inv, det.

eigfact \((A, B)\)
Compute the generalized eigenvalue decomposition of \(A\) and \(B\) and return an \text{GeneralizedEigen} object. If \(F\) is the factorization object, the eigenvalues can be accessed with \(F[:values]\) and the eigenvectors with \(F[:vectors]\).

eigfact! \((A[, \text{B}])\)
eigfact! is the 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 \text{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 \texttt{HessenbergQ} object, and may be converted to a regular matrix with \texttt{full}.

**\texttt{hessfact!}(A)**

\texttt{hessfact!} is the same as \texttt{hessfact()}, but saves space by overwriting the input \(A\), instead of creating a copy.

**\texttt{schurfact}(A) \rightarrow \text{Schur}**

Computes the Schur factorization of the matrix \(A\). The (quasi) triangular Schur factor can be obtained from the \texttt{Schur} object \(F\) with either \(F[:\text{Schur}]\) or \(F[:T]\) and the unitary/orthogonal Schur vectors can be obtained with \(F[:\text{vectors}]\) or \(F[:Z]\) such that \(A=F[:\text{vectors}]*F[:\text{Schur}]*F[:\text{vectors}]'\). The eigenvalues of \(A\) can be obtained with \(F[:\text{values}]\).

**\texttt{schurfact!}(A)\rightarrow \text{Schur}**

Computer the Schur factorization of \(A\), overwriting \(A\) in the process. See \texttt{schurfact()}.

**\texttt{schur}(A) \rightarrow \text{Schur}[:T], \text{Schur}[:Z], \text{Schur}[:\text{values}]**

See \texttt{schurfact}

**\texttt{schurfact}(A, B) \rightarrow \text{GeneralizedSchur}**

Computes the Generalized Schur (or QZ) factorization of the matrices \(A\) and \(B\). The (quasi) triangular Schur factors can be obtained from the \texttt{Schur} object \(F\) with \(F[:S]\) and \(F[:T]\), the left unitary/orthogonal Schur vectors can be obtained with \(F[:\text{left}]\) or \(F[:Q]\) and the right unitary/orthogonal Schur vectors can be obtained with \(F[:\text{right}]\) or \(F[:Z]\) such that \(A=F[:\text{left}]*F[:S]*F[:\text{right}]'\) and \(B=F[:\text{left}]*F[:T]*F[:\text{right}]'\). The generalized eigenvalues of \(A\) and \(B\) can be obtained with \(F[:\alpha]/F[:\beta]\).

**\texttt{schur}(A, B) \rightarrow \text{GeneralizedSchur}[:S], \text{GeneralizedSchur}[:T], \text{GeneralizedSchur}[:Q], \text{GeneralizedSchur}[:Z]**

See \texttt{schurfact}

**\texttt{svdfact}(A[, thin]) \rightarrow \text{SVD}**

Compute the Singular Value Decomposition (SVD) of \(A\) and return an \texttt{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 \(\text{thin} = \text{true}\), 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.

**\texttt{svdfact!}(A[, thin]) \rightarrow \text{SVD}**

\texttt{svdfact!} is the same as \texttt{svdfact()}, but saves space by overwriting the input \(A\), instead of creating a copy. If \(\text{thin} = \text{true}\), an economy mode decomposition is returned. The default is to produce a thin decomposition.

**\texttt{svd}(A[, thin]) \rightarrow U, S, V**

Compute the SVD of \(A\), returning \(U\), vector \(S\), and \(V\) such that \(A = U*\text{diagm}(S) * V'\). If \(\text{thin} = \text{true}\), an economy mode decomposition is returned.

**\texttt{svdvals}(A)**

Returns the singular values of \(A\).

**\texttt{svdvals!}(A)**

Returns the singular values of \(A\), while saving space by overwriting the input.

**\texttt{svdfact}(A, B) \rightarrow \text{GeneralizedSVD}**

Compute the generalized SVD of \(A\) and \(B\), returning a \texttt{GeneralizedSVD} Factorization object, such that \(A = U*D1*R0*Q'\) and \(B = V*D2*R0*Q'\).

**\texttt{svd}(A, B) \rightarrow U, V, Q, D1, D2, R0**

Compute the generalized SVD of \(A\) and \(B\), returning \(U, V, Q, D1, D2\), and \(R0\) such that \(A = U*D1*R0*Q'\) and \(B = V*D2*R0*Q'\).

**\texttt{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.

diagm\((v[, k])\)
Construct a diagonal matrix and place v on the k-th diagonal.

scale\((A, B)\)
scale\((A::Array, B::Number)\) scales all values in A with B. Note: In cases where the array is big enough, scale can be much faster than A .* B, due to the use of BLAS.

scale\((A::Matrix, B::Vector)\) is the same as multiplying with a diagonal matrix on the right, and scales the columns of A with the values in B.

scale\((A::Vector, B::Matrix)\) is the same as multiplying with a diagonal matrix on the left, and scales the rows of B with the values in A.

scale\(!\((A, B)\)
scale\(!\((A, B)\) overwrites the input array with the scaled result.

symmetrize\(!\((A[, UL::Char])\)
symmetrize\(!\((A)\) converts from the BLAS/LAPACK symmetric storage format, in which only the UL (’U’pper or ‘L’ower, default ‘U’) triangle is used, to a full symmetric matrix.

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 a matrix. p is 2 by default, if not provided. If A is a vector, norm\((A, p)\) computes the p-norm. norm\((A, \inf)\) returns the largest value in abs\((A)\), whereas norm\((A, -\inf)\)
returns the smallest. If \( A \) is a matrix, valid values for \( p \) are 1, 2, or \( \text{Inf} \). In order to compute the Frobenius norm, use \text{normfro}.

\[ \text{normfro}(A) \]
Compute the Frobenius norm of a matrix \( A \).

\[ \text{cond}(M[p]) \]
Matrix condition number, computed using the p-norm. \( p \) is 2 by default, if not provided. Valid values for \( p \) are 1, 2, or \( \text{Inf} \).

\[ \text{trace}(M) \]
Matrix trace

\[ \text{det}(M) \]
Matrix determinant

\[ \text{logdet}(M) \]
Log of Matrix determinant. Equivalent to \( \log(\text{det}(M)) \), but may provide increased accuracy and/or speed.

\[ \text{inv}(M) \]
Matrix inverse

\[ \text{pinv}(M) \]
Moore-Penrose inverse

\[ \text{null}(M) \]
Basis for null space of \( M \).

\[ \text{repmat}(A, n, m) \]
Construct a matrix by repeating the given matrix \( n \) times in dimension 1 and \( m \) times in dimension 2.

\[ \text{repeat}(A, \text{inner} = \text{Int}[], \text{outer} = \text{Int}[]) \]
Construct an array by repeating the entries of \( A \). The i-th element of \( \text{inner} \) specifies the number of times that the individual entries of the i-th dimension of \( A \) should be repeated. The i-th element of \( \text{outer} \) specifies the number of times that a slice along the i-th dimension of \( A \) should be repeated.

\[ \text{kron}(A, B) \]
Kronecker tensor product of two vectors or two matrices.

\[ \text{linreg}(x, y) \]
Determine parameters \([a, b]\) that minimize the squared error between \( y \) and \( a+b\cdot x \).

\[ \text{linreg}(x, y, w) \]
Weighted least-squares linear regression.

\[ \text{expm}(A) \]
Matrix exponential.

\[ \text{issym}(A) \]
Test whether a matrix is symmetric.

\[ \text{isposdef}(A) \]
Test whether a matrix is positive-definite.

\[ \text{isposdef!}(A) \]
Test whether a matrix is positive-definite, overwriting \( A \) in the processes.

\[ \text{istril}(A) \]
Test whether a matrix is lower-triangular.

\[ \text{istriu}(A) \]
Test whether a matrix is upper-triangular.
ishermitian $(A)$
Test whether a matrix is hermitian.

transpose $(A)$
The transpose operator (‘).  

ctranspose $(A)$
The conjugate transpose operator (’).  

eigs $(A; nev=6, which="LM", tol=0.0, maxiter=1000, sigma=0, ritzvec=true, op_part=:real, v0=zeros((0,)))$ -> $(d, v, nconv, niter, nmult, resid)$

eigs computes eigenvalues $d$ of $A$ using Arnoldi factorization. The following keyword arguments are supported:

- **nev**: Number of eigenvalues
- **which**: type of eigenvalues (“LM”, “SM”)
- **tol**: tolerance ($tol \leq 0.0$ defaults to $DLAMCH(’EPS$))
- **maxiter**: Maximum number of iterations
- **sigma**: find eigenvalues close to $sigma$ using shift and invert
- **ritzvec**: Returns the Ritz vectors $v$ (eigenvectors) if true
- **op_part**: which part of linear operator to use for real $A$ (:real, :imag)
- **v0**: starting vector from which to start the Arnoldi iteration

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 $resid$.

svds $(A; nev=6, which="LA", tol=0.0, maxiter=1000, ritzvec=true)$

svds computes the singular values of $A$ using Arnoldi factorization. The following keyword arguments are supported:

- **nsv**: Number of singular values
- **which**: type of singular values (“LA”)
- **tol**: tolerance ($tol \leq 0.0$ defaults to $DLAMCH(’EPS$))
- **maxiter**: Maximum number of iterations
- **ritzvec**: Returns the singular vectors if true

peakflops $(n; parallel=false)$
peakflops computes the peak flop rate of the computer by using BLAS dgemm. By default, if no arguments are specified, it multiplies a matrix of size $n \times n$, 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.

### 2.1.43 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** $(n, X, inx, Y, incy)$
Dot product of two vectors consisting of $n$ elements of array $X$ with stride $inx$ and $n$ elements of array $Y$ with stride $incy$. There are no `dot` methods for Complex arrays.

The following functions are defined within the `Base.LinAlg.BLAS` module.

**blascopy!** $(n, X, inx, Y, incy)$
Copy $n$ elements of array $X$ with stride $inx$ to array $Y$ with stride $incy$. Returns $Y$.

**nrm2** $(n, X, inx)$
2-norm of a vector consisting of $n$ elements of array $X$ with stride $inx$.

**asum** $(n, X, inx)$
sum of the absolute values of the first $n$ elements of array $X$ with stride $inx$.

**axpy!** $(n, a, X, inx, Y, incy)$
Overwrite $Y$ with $a \cdot X + Y$. Returns $Y$.

**scal!** $(n, a, X, inx)$
Overwrite $X$ with $a \cdot X$. Returns $X$.

**scal** $(n, a, X, inx)$
Returns $a \cdot X$.

**syrk** $(uplo, trans, alpha, A, beta, C)$
Rank-k update of the symmetric matrix $C$ as $\alpha \cdot A \cdot A' + \beta \cdot C$ or $\alpha \cdot A' \cdot A + \beta \cdot C$ 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 $\alpha \cdot A \cdot A'$ or $\alpha \cdot A' \cdot A$, 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 $\alpha \cdot A \cdot A' + \beta \cdot C$ or $\alpha \cdot A' \cdot A + \beta \cdot C$ 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 $\alpha \cdot A \cdot A'$ or $\alpha \cdot A' \cdot A$, according to $trans$ (‘N’ or ‘T’).

**gbmv** $(trans, m, kl, ku, alpha, A, x, beta, y)$
Update vector $y$ as $\alpha \cdot A \cdot x + \beta \cdot y$ or $\alpha \cdot A' \cdot x + \beta \cdot y$ according to $trans$ (‘N’ or ‘T’). The matrix $A$ is a general band matrix of dimension $m$ by `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 $\alpha \cdot A \cdot x$ or $\alpha \cdot A' \cdot x$ according to $trans$ (‘N’ or ‘T’). The matrix $A$ is a general band matrix of dimension $m$ by `size(A, 2)` with $kl$ sub-diagonals and $ku$ super-diagonals.

**smbv** $(uplo, k, alpha, A, x, beta, y)$
Update vector $y$ as $\alpha \cdot A \cdot x + \beta \cdot y$ where $A$ is a symmetric band matrix of order `size(A, 2)` with $k$ super-diagonals stored in the argument $A$. The storage layout for $A$ is described the reference BLAS module, level-2 BLAS at [http://www.netlib.org/lapack/explore-html/](http://www.netlib.org/lapack/explore-html/).

Returns the updated $y$. 
The Julia Standard Library

238  Chapter 2. The Julia Standard Library

\textbf{sbmv (uplo, k, alpha, A, x)}

Returns \(\alpha A^\ast x\) where \(A\) is a symmetric band matrix of order \(\text{size}(A, 2)\) with \(k\) super-diagonals stored in the argument \(A\).

\textbf{sbmv (uplo, k, A, x)}

Returns \(A^\ast x\) where \(A\) is a symmetric band matrix of order \(\text{size}(A, 2)\) with \(k\) super-diagonals stored in the argument \(A\).

\textbf{gemm! (tA, tB, alpha, A, B, beta, C)}

Update \(C\) as \(\alpha A^\ast B + \beta C\) or the other three variants according to \(tA\) (transpose \(A\)) and \(tB\). Returns the updated \(C\).

\textbf{gemm (tA, tB, alpha, A, B)}

Returns \(\alpha A^\ast B\) or the other three variants according to \(tA\) (transpose \(A\)) and \(tB\).

\textbf{gamm (tA, tB, alpha, A, B)}

Returns \(\alpha A^\ast B\) or the other three variants according to \(tA\) (transpose \(A\)) and \(tB\).

\textbf{gemv! (tA, alpha, A, x, beta, y)}

Update the vector \(y\) as \(\alpha A^\ast x + \beta y\) or \(\alpha A' x + \beta y\) according to \(tA\) (transpose \(A\)). Returns the updated \(y\).

\textbf{gemv (tA, alpha, A, x)}

Returns \(\alpha A^\ast x\) or \(\alpha A' x\) according to \(tA\) (transpose \(A\)).

\textbf{gemv (tA, alpha, A, x)}

Returns \(\alpha A^\ast x\) or \(\alpha A' x\) according to \(tA\) (transpose \(A\)).

\textbf{symm! (side, ul, alpha, A, B, beta, C)}

Update \(C\) as \(\alpha A^\ast B + \beta C\) or \(\alpha B^\ast A + \beta C\) according to \(\text{side}\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used. Returns the updated \(C\).

\textbf{symm (side, ul, alpha, A, B)}

Returns \(\alpha A^\ast B\) or \(\alpha B^\ast A\) according to \(\text{side}\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used.

\textbf{symm (side, ul, A, B)}

Returns \(A^\ast B\) or \(B^\ast A\) according to \(\text{side}\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used.

\textbf{symm (tA, tB, alpha, A, B)}

Returns \(\alpha A^\ast B\) or the other three variants according to \(tA\) (transpose \(A\)) and \(tB\).

\textbf{symm! (ul, alpha, A, x, beta, y)}

Update the vector \(y\) as \(\alpha A^\ast y + \beta y\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used. Returns the updated \(y\).

\textbf{symm (ul, alpha, A, x)}

Returns \(\alpha A^\ast x\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used.

\textbf{symm (ul, A, x)}

Returns \(A^\ast x\). \(A\) is assumed to be symmetric. Only the \(ul\) triangle of \(A\) is used.

\textbf{trmm! (side, ul, tA, dA, alpha, A, B)}

Update \(B\) as \(\alpha A^\ast B\) or one of the other three variants determined by \(\text{side}\) (\(A\) 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 \(B\).

\textbf{trmm (side, ul, tA, dA, alpha, A, B)}

Returns \(\alpha A^\ast B\) or one of the other three variants determined by \(\text{side}\) (\(A\) 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).
\textbf{trsm!} (side, ul, tA, dA, alpha, A, B)

Overwrite B with the solution to \(A \times X = \alpha \times B\) 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.

\textbf{trsm} (side, ul, tA, dA, alpha, A, B)

Returns the solution to \(A \times X = \alpha \times B\) 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).

\textbf{trmv!} (side, ul, tA, dA, alpha, A, b)

Update b as \(\alpha \times A \times b\) 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.

\textbf{trmv} (side, ul, tA, dA, alpha, A, b)

Returns \(\alpha \times A \times b\) 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).

\textbf{trsv!} (side, ul, tA, dA, alpha, A, b)

Overwrite b with the solution to \(A \times X = \alpha \times b\) 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.

\textbf{trsv} (side, ul, tA, dA, alpha, A, b)

Returns the solution to \(A \times X = \alpha \times b\) 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).

\textbf{blas_set_num_threads} (n)

Set the number of threads the BLAS library should use.

\section*{2.1.44 Constants}

\textbf{OS_NAME}

A symbol representing the name of the operating system. Possible values are :\texttt{Linux}, :\texttt{Darwin} (OS X), or :\texttt{Windows}.

\textbf{ARGS}

An array of the command line arguments passed to Julia, as strings.

\textbf{C_NULL}

The C null pointer constant, sometimes used when calling external code.

\textbf{CPU_CORES}

The number of CPU cores in the system.

\textbf{WORD_SIZE}

Standard word size on the current machine, in bits.

\textbf{VERSION}

An object describing which version of Julia is in use.

\textbf{LOAD_PATH}

An array of paths (as strings) where the \texttt{require} function looks for code.
2.1.45 Filesystem

`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.

`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.

tempname()
   Generate a unique temporary filename.

tempdir()
   Obtain the path of a temporary directory.

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.

### 2.1.46 Punctuation

Extended documentation for mathematical symbols & functions is [here](#).

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>@m</td>
<td>invoke macro m; followed by space-separated expressions</td>
</tr>
<tr>
<td>!</td>
<td>prefix “not” operator</td>
</tr>
<tr>
<td>a!{ }</td>
<td>at the end of a function name, ! indicates that a function modifies its argument(s)</td>
</tr>
<tr>
<td>#</td>
<td>begin single line comment</td>
</tr>
<tr>
<td>$</td>
<td>xor operator, string and expression interpolation</td>
</tr>
<tr>
<td>%</td>
<td>remainder operator</td>
</tr>
<tr>
<td>^</td>
<td>exponent operator</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td>*</td>
<td>multiply, or matrix multiply</td>
</tr>
<tr>
<td>()</td>
<td>the empty tuple</td>
</tr>
<tr>
<td>~</td>
<td>bitwise not operator</td>
</tr>
<tr>
<td>\</td>
<td>backslash operator</td>
</tr>
<tr>
<td>,</td>
<td>complex transpose operator $A^H$</td>
</tr>
<tr>
<td>a[]</td>
<td>array indexing</td>
</tr>
<tr>
<td>[, ]</td>
<td>vertical concatenation</td>
</tr>
<tr>
<td>[;;]</td>
<td>also vertical concatenation</td>
</tr>
<tr>
<td>[ ]</td>
<td>with space-separated expressions, horizontal concatenation</td>
</tr>
<tr>
<td>T{ }</td>
<td>parametric type instantiation</td>
</tr>
<tr>
<td>{}</td>
<td>construct a cell array</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.1 – continued from previous page

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>statement separator</td>
</tr>
<tr>
<td>,</td>
<td>separate function arguments or tuple components</td>
</tr>
<tr>
<td>?</td>
<td>3-argument conditional operator (conditional ? if_true : if_false)</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>delimit string literals</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>delimit character literals</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>delimit external process (command) specifications</td>
</tr>
<tr>
<td>...</td>
<td>splice arguments into a function call or declare a varargs function or type</td>
</tr>
<tr>
<td>.</td>
<td>access named fields in objects or names inside modules, also prefixes elementwise operators</td>
</tr>
<tr>
<td>a:b</td>
<td>range a, a+1, a+2, ..., b</td>
</tr>
<tr>
<td>a:s:b</td>
<td>range a, a+s, a+2s, ..., b</td>
</tr>
<tr>
<td>:</td>
<td>index an entire dimension (1:end)</td>
</tr>
<tr>
<td>::</td>
<td>type annotation, depending on context</td>
</tr>
<tr>
<td>:( )</td>
<td>quoted expression</td>
</tr>
</tbody>
</table>

2.1.47 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 Int64 Array:
  1
  2
  3

You can easily sort in reverse order as well:

julia> sort([2,3,1], rev=true)
3-element Int64 Array:
  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 Int64 Array:
  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 Float64 Array:
  0.587746
  -0.870797
  0.870797
Arrays can easily be sorted according to an arbitrary transformation of their values:

```julia
julia> sort(v, by=abs)
5-element `Float64` Array:
-0.111843
0.587746
-0.870797
1.08793
-1.25061
```

Or in reverse order by a transformation:

```julia
julia> sort(v, by=abs, rev=true)
5-element `Float64` Array:
-1.25061
1.08793
-0.870797
0.587746
-0.111843
```

Reasonable sorting algorithms are used by default, but you can choose other algorithms as well:

```julia
julia> sort(v, alg=InsertionSort)
5-element `Float64` Array:
-1.25061
-0.870797
-0.111843
0.587746
1.08793
```

### Sorting Functions

`sort!(v, [dim,] [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 \texttt{lt} function is applied to the result of the \texttt{by} function; \texttt{rev=true} reverses whatever ordering specified via the \texttt{by} and \texttt{lt} keywords.

\texttt{sort (v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])}

Variant of \texttt{sort!} that returns a sorted copy of \texttt{v} leaving \texttt{v} itself unmodified.

\texttt{sort (A, dim, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])}

Sort a multidimensional array \texttt{A} along the given dimension.

\texttt{sortperm (v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])}

Return a permutation vector of indices of \texttt{v} that puts it in sorted order. Specify \texttt{alg} to choose a particular sorting algorithm (see Sorting Algorithms). \texttt{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 \texttt{QuickSort}, a different permutation that puts the array into order may be returned. The order is specified using the same keywords as \texttt{sort!}.

\texttt{sortrows (A, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])}

Sort the rows of matrix \texttt{A} lexicographically.

\texttt{sortcols (A, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])}

Sort the columns of matrix \texttt{A} lexicographically.

Order-Related Functions

\texttt{issorted (v, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Test whether a vector is in sorted order. The \texttt{by}, \texttt{lt} and \texttt{rev} keywords modify what order is considered to be sorted just as they do for \texttt{sort}.

\texttt{searchsorted (a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Returns the range of indices of \texttt{a} which compare as equal to \texttt{x} according to the order specified by the \texttt{by}, \texttt{lt} and \texttt{rev} keywords, assuming that \texttt{a} is already sorted in that order. Returns an empty range located at the insertion point if \texttt{a} does not contain values equal to \texttt{x}.

\texttt{searchsortedfirst (a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Returns the index of the first value in \texttt{a} greater than or equal to \texttt{x}, according to the specified order. Returns \texttt{length(a)+1} if \texttt{x} is greater than all values in \texttt{a}.

\texttt{searchsortedlast (a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Returns the index of the last value in \texttt{a} less than or equal to \texttt{x}, according to the specified order. Returns \texttt{0} if \texttt{x} is less than all values in \texttt{a}.

\texttt{select! (v, k, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Partially sort the vector \texttt{v} in place, according to the order specified by \texttt{by}, \texttt{lt} and \texttt{rev} so that the value at index \texttt{k} (or range of adjacent values if \texttt{k} is a range) occurs at the position where it would appear if the array were fully sorted. If \texttt{k} is a single index, that values is returned; if \texttt{k} is a range, an array of values at those indices is returned. Note that \texttt{select!} does not fully sort the input array, but does leave the returned elements where they would be if the array were fully sorted.

\texttt{select (v, k, [by=<transform>,] [lt=<comparison>,] [rev=false])}

Variant of \texttt{select!} which copies \texttt{v} before partially sorting it, thereby returning the same thing as \texttt{select!} but leaving \texttt{v} unmodified.

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 \texttt{alg=<algorithm>} as a keyword argument after the array to be sorted.

### 2.2 Built-in Modules

#### 2.2.1 Package Manager Functions

All Package manager functions are defined in the \texttt{Pkg} module; note that none of the \texttt{Pkg} module’s functions are exported; to use them, you’ll need to prefix each function call with an explicit \texttt{Pkg.}, e.g. \texttt{Pkg.status()} or \texttt{Pkg.dir()}.

\textbf{dir()} \rightarrow \text{String}

Returns the absolute path of the package directory. This defaults to \texttt{joinpath(homedir(),".julia")} on all platforms (i.e. \texttt{~/.julia} in UNIX shell syntax). If the \texttt{JULIA_PKGDIR} environment variable is set, that path is used instead. If \texttt{JULIA_PKGDIR} is a relative path, it is interpreted relative to whatever the current working directory is.

\textbf{dir(names...)} \rightarrow \text{String}

Equivalent to \texttt{normpath(Pkg.dir(),names...)} – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, \texttt{Pkg.dir(pkg)} returns the path to the package \texttt{pkg}.

\textbf{init()}

Initialize \texttt{Pkg.dir()} as a package directory. This will be done automatically when the \texttt{JULIA_PKGDIR} is not set and \texttt{Pkg.dir()} uses its default value.

\textbf{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 \texttt{Pkg.dir("REQUIRE")} and the state of installed packages in \texttt{Pkg.dir()}, Packages that are no longer required are moved into \texttt{Pkg.dir(".trash")}.

\textbf{edit()}

Opens \texttt{Pkg.dir("REQUIRE")} in the editor specified by the \texttt{VISUAL} or \texttt{EDITOR} environment variables; when the editor command returns, it runs \texttt{Pkg.resolve()} to determine and install a new optimal set of installed package versions.

\textbf{add(pkg, vers...)}

Add a requirement entry for \texttt{pkg} to \texttt{Pkg.dir("REQUIRE")} and call \texttt{Pkg.resolve()}. If \texttt{vers} are given, they must be \texttt{VersionNumber} objects and they specify acceptable version intervals for \texttt{pkg}.

\textbf{rm(pkg)}

Remove all requirement entries for \texttt{pkg} from \texttt{Pkg.dir("REQUIRE")} and call \texttt{Pkg.resolve()}. 245
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.

pin(pkg)
Pin pkg at the current version.

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 scripts 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.
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, push METADATA.

### 2.2.2 Collections and Data Structures

The Collections module contains implementations of some common data structures.

**PriorityQueue**

The PriorityQueue type is 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 = 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
```

**Heap Functions**

Along with the PriorityQueue type are 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! \((v[\cdot, \text{ord}])\)
Given a binary heap-ordered array, push a new element, preserving the heap property. For efficiency, this function does not check that the array is indeed heap-ordered.

heappop! \((v[\cdot, \text{ord}])\)
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.

### 2.2.3 Graphics

The \texttt{Base.Graphics} interface is an abstract wrapper; specific packages (e.g., Cairo and Tk/Gtk) implement much of the functionality.

#### Geometry

\texttt{Vec2}(x, y)
Creates a point in two dimensions

\texttt{BoundingBox}(xmin, xmax, ymin, ymax)
Creates a box in two dimensions with the given edges

\texttt{BoundingBox}(\texttt{objs}...)
Creates a box in two dimensions that encloses all objects

\texttt{width}(\texttt{obj})
Computes the width of an object

\texttt{height}(\texttt{obj})
Computes the height of an object

\texttt{xmin}(\texttt{obj})
Computes the minimum x-coordinate contained in an object

\texttt{xmax}(\texttt{obj})
Computes the maximum x-coordinate contained in an object

\texttt{ymin}(\texttt{obj})
Computes the minimum y-coordinate contained in an object

\texttt{ymax}(\texttt{obj})
Computes the maximum y-coordinate contained in an object

\texttt{diagonal}(\texttt{obj})
Return the length of the diagonal of an object

\texttt{aspect\_ratio}(\texttt{obj})
Compute the height/width of an object

\texttt{center}(\texttt{obj})
Return the point in the center of an object

\texttt{xrange}(\texttt{obj})
Returns a tuple \((\texttt{xmin(\texttt{obj})}, \texttt{xmax(\texttt{obj})})\)

\texttt{yrange}(\texttt{obj})
Returns a tuple \((\texttt{ymin(\texttt{obj})}, \texttt{ymax(\texttt{obj})})\)

\texttt{rotate}(\texttt{obj, angle, origin}) \rightarrow \texttt{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

**scale** *(obj, s::Real)*

Scale the width and height of a graphics object, keeping the center fixed

**add** *(bb1::BoundingBox, bb2::BoundingBox) → BoundingBox*

Returns the smallest box containing both boxes

**and** *(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

### 2.2.4 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.

**Overview**

To use the default handler, the macro `@test()` can be used directly:

```julia
# Julia code
julia> @test 1 == 1
julia> @test 1 == 0
ERROR: test failed: :((1==0))
in default_handler at test.jl:20
in do_test at test.jl:37

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:35
```

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 error, `@test_throws()`:

```julia
julia> @test_throws error("An error")
```

### 2.2. Built-in Modules
As floating point comparisons can be imprecise, two additional macros exist taking in account small numerical errors:

```
julia> @test_approx_eq 1. 0.9999999999999
ERROR: assertion failed: |1.0 - 0.9999999999| < -0.2817181715409549
  1.0 = 1.0
  0.999 = 0.999
in test_approx_eq at test.jl:75
in test_approx_eq at test.jl:80
```

```
julia> @test_approx_eq_eps 1. 0.999 e-2
julia> @test_approx_eq_eps 1. 0.999 e-3
ERROR: assertion failed: |1.0 - 0.999| < -0.2817181715409549
  1.0 = 1.0
  0.999 = 0.999
in test_approx_eq at test.jl:75
```

### Handlers

A handler is a function defined for three kinds of arguments: `Success`, `Failure`, `Error`:

```julia
# 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
```
Macros

@test (ex)
Test the expression ex and calls the current handler to handle the result.

@test_throws (ex)
Test the expression ex and calls the current handler to handle the result in the following manner:
  • If the test doesn’t throw an error, the Failure case is called.
  • If the test throws an error, the Success case is called.

@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.

Functions

with_handler (f, handler)
Run the function f using the handler as the handler.

2.2.5 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). However, this design has important 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 will run at nearly native speed.

For these reasons, it’s recommended that you try using the built-in sampling profiler before considering any alternatives.

Basic usage

Let’s work with a simple test case:
function myfunc()
    A = rand(100, 100, 200)
    max(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()

Now let’s see the results:

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 librandom.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 librandom.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
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
```

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. Below you’ll see how you can detect such events.

This illustrates the default “tree” dump; an alternative is the “flat” dump, which accumulates counts independent of their nesting:

```julia
julia> Profile.print(format=:flat)
```

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:

```julia
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:

```julia
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.
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().

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:

```julia
function print(io::IO = STDOUT, data = fetch(); format = :tree, C = false, combine = true, cols = tty_cols())
```

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:

```julia
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:

```julia
33 inference.jl; abstract_call; line: 645
33 ...rence.jl; abstract_call_GF; line: 567
33 ...nce.jl; typeinf; line: 1201
+1 5 ...nce.jl; ...nterpret; line: 900
+3 5 ...nce.jl; abstract_eval; line: 758
+4 5 ...nce.jl; ...ct_eval_call; line: 733
+6 5 ...nce.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:

```julia
s = open("/tmp/prof.txt","w")
Profile.print(s,cols = 500)
close(s)
```
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 configure the relevant parameters this way:

Profile.init(n, delay)

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).

Function reference

@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 Profile.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 are n=10^6 and delay=0.001.

fetch() → data

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like Profile.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; Profile.retrieve() may be a better choice for most users.

retrieve(;C = false) → 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.