I/O and Network

General I/O

`Base.stdout` — Constant

```
stdout
```

Global variable referring to the standard out stream.

`Base.stderr` — Constant

```
stderr
```

Global variable referring to the standard error stream.

`Base.stdin` — Constant

```
stdin
```

Global variable referring to the standard input stream.

`Base.open` — Function

```
open(f::Function, args...; kwargs....)
```

Apply the function `f` to the result of `open(args...; kwargs...)` and close the resulting file descriptor upon completion.

Examples

```
julia> open("myfile.txt", "w") do io
           write(io, "Hello world!")
       end;

julia> open(f->read(f, String), "myfile.txt")
"Hello world!"

julia> rm("myfile.txt")
```

```
open(filename::AbstractString; lock = true, keywords...) -> IOStream
```

Open a file in a mode specified by five boolean keyword arguments:

Keyword	Description	Default
`read`	open for reading	`!write`
`write`	open for writing	`truncate | append`
`create`	create if non-existent	`!read & write | truncate | append`
`truncate`	truncate to zero size	`!read & write`
`append`	seek to end	`false`

The default when no keywords are passed is to open files for reading only. Returns a stream for accessing the opened file.

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Julia 1.5

The `lock` argument is available as of Julia 1.5.

```
open(filename::AbstractString, [mode::AbstractString]; lock = true) -> IOStream
```

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

Mode	Description	Keywords
`r`	read	none
`w`	write, create, truncate	`write = true`
`a`	write, create, append	`append = true`
`r+`	read, write	`read = true, write = true`
`w+`	read, write, create, truncate	`truncate = true, read = true`
`a+`	read, write, create, append	`append = true, read = true`

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Examples

```
julia> io = open("myfile.txt", "w");

julia> write(io, "Hello world!");

julia> close(io);

julia> io = open("myfile.txt", "r");

julia> read(io, String)
"Hello world!"

julia> write(io, "This file is read only")
ERROR: ArgumentError: write failed, IOStream is not writeable
[...]

julia> close(io)

julia> io = open("myfile.txt", "a");

julia> write(io, "This stream is not read only")
28

julia> close(io)
julia> rm("myfile.txt")
```

Julia 1.5

The `lock` argument is available as of Julia 1.5.

```
open(fd::OS_HANDLE) -> IO
```

Take a raw file descriptor wrap it in a Julia-aware IO type, and take ownership of the fd handle. Call `open(Libc.dup(fd))` to avoid the ownership capture of the original handle.

Warn

Do not call this on a handle that's already owned by some other part of the system.

```
open(command, mode::AbstractString, stdio=devnull)
```

Run `command` asynchronously. Like `open(command, stdio; read, write)` except specifying the read and write flags via a mode string instead of keyword arguments. Possible mode strings are:

Mode	Description	Keywords
`r`	read	none
`w`	write	`write = true`
`r+`	read, write	`read = true, write = true`
`w+`	read, write	`read = true, write = true`

```
open(command, stdio=devnull; write::Bool = false, read::Bool = !write)
```

Start running `command` asynchronously, and return a `process::IO` object. If `read` is true, then reads from the process come from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `write` is true, then writes go to the process's standard input and `stdio` optionally specifies the process's standard output stream. The process's standard error stream is connected to the current global `stderr`.

```
open(f::Function, command, args...; kwargs...)
```

Similar to `open(command, args...; kwargs...)`, but calls `f(stream)` on the resulting process stream, then closes the input stream and waits for the process to complete. Returns the value returned by `f`.

`Base.IOStream` — Type

```
IOStream
```

A buffered IO stream wrapping an OS file descriptor. Mostly used to represent files returned by `open`.

`Base.IOBuffer` — Type

```
IOBuffer([data::AbstractVector{UInt8}]; keywords...) -> IOBuffer
```

Create an in-memory I/O stream, which may optionally operate on a pre-existing array.

It may take optional keyword arguments:

- `read`, `write`, `append`: restricts operations to the buffer; see `open` for details.
- `truncate`: truncates the buffer size to zero length.
- `maxsize`: specifies a size beyond which the buffer may not be grown.
- `sizehint`: suggests a capacity of the buffer (`data` must implement `sizehint!(data, size)`).

When `data` is not given, the buffer will be both readable and writable by default.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members."
56
```

```
julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> io = IOBuffer(b"JuliaLang is a GitHub organization.")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=

julia> read(io, String)
"JuliaLang is a GitHub organization."

julia> write(io, "This isn't writable.")
ERROR: ArgumentError: ensureroom failed, IOBuffer is not writeable

julia> io = IOBuffer(UInt8[], read=true, write=true, maxsize=34)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=f

julia> write(io, "JuliaLang is a GitHub organization.")
34

julia> String(take!(io))
"JuliaLang is a GitHub organization"

julia> length(read(IOBuffer(b"data", read=true, truncate=false)))
4

julia> length(read(IOBuffer(b"data", read=true, truncate=true)))
0
```

```
IOBuffer(string::String)
```

Create a read-only `IOBuffer` on the data underlying the given string.

Examples

```
julia> io = IOBuffer("Haho");

julia> String(take!(io))
"Haho"

julia> String(take!(io))
"Haho"
```

`Base.take!` — Method

```
take!(b::IOBuffer)
```

Obtain the contents of an `IOBuffer` as an array, without copying. Afterwards, the `IOBuffer` is reset to its initial state.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."
```

`Base.fdio` — Function

```
fdio([name::AbstractString, ]fd::Integer[, own::Bool=false]) -> IOStream
```

Create an `IOStream` object from an integer file descriptor. If `own` is `true`, closing this object will close the underlying descriptor. By default, an `IOStream` is closed when it is garbage collected. `name` allows you to associate the descriptor with a named file.

`Base.flush` — Function

```
flush(stream)
```

Commit all currently buffered writes to the given stream.

`Base.close` — Function

```
close(stream)
```

Close an I/O stream. Performs a `flush` first.

`Base.write` — Function

```
write(io::IO, x)
write(filename::AbstractString, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Return the number of bytes written into the stream. See also `print` to write a text representation (with an encoding that may depend upon `io`).

The endianness of the written value depends on the endianness of the host system. Convert to/from a fixed endianness when writing/reading (e.g. using `htol` and `ltoh`) to get results that are consistent across platforms.

You can write multiple values with the same `write` call. i.e. the following are equivalent:

```
write(io, x, y...)
write(io, x) + write(io, y...)
```

Examples

Consistent serialization:

```
julia> fname = tempname(); # random temporary filename

julia> open(fname,"w") do f
           # Make sure we write 64bit integer in little-endian byte order
           write(f,htol(Int64(42)))
       end
8

julia> open(fname,"r") do f
           # Convert back to host byte order and host integer type
           Int(ltoh(read(f,Int64)))
       end
42
```

Merging write calls:

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members."
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> write(io, "Sometimes those members") + write(io, " write documentation."
44

julia> String(take!(io))
"Sometimes those members write documentation."
```

User-defined plain-data types without `write` methods can be written when wrapped in a `Ref`:

```
julia> struct MyStruct; x::Float64; end

julia> io = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=f

julia> write(io, Ref(MyStruct(42.0)))
8

julia> seekstart(io); read!(io, Ref(MyStruct(NaN)))
Base.RefValue{MyStruct}(MyStruct(42.0))
```

`Base.read` — Function

```
read(io::IO, T)
```

Read a single value of type `T` from `io`, in canonical binary representation.

Note that Julia does not convert the endianness for you. Use `ntoh` or `ltoh` for this purpose.

```
read(io::IO, String)
```

Read the entirety of `io`, as a `String`.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, String)
"JuliaLang is a GitHub organization"
```

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

```
read(s::IO, nb=typemax(Int))
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

```
read(s::IOStream, nb::Integer; all=true)
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

```
read(command::Cmd)
```

Run `command` and return the resulting output as an array of bytes.

```
read(command::Cmd, String)
```

Run `command` and return the resulting output as a `String`.

`Base.read!` — Function

```
read!(stream::IO, array::AbstractArray)
read!(filename::AbstractString, array::AbstractArray)
```

Read binary data from an I/O stream or file, filling in `array`.

`Base.readbytes!` — Function

```
readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

```
readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=length(b); all::Bool=t
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

`Base.unsafe_read` — Function

```
unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from the `IO` stream object into `ref` (converted to a pointer).

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_read(s::T, p::Ptr{UInt8}, n::UInt)`

`Base.unsafe_write` — Function

```
unsafe_write(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from `ref` (converted to a pointer) into the `IO` object.

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_write(s::T, p::Ptr{UInt8}, n::UInt)`

`Base.peek` — Function

```
peek(stream[, T=UInt8])
```

Read and return a value of type `T` from a stream without advancing the current position in the stream.

Examples

```
julia> b = IOBuffer("julia");

julia> peek(b)
0x6a

julia> position(b)
0

julia> peek(b, Char)
'j': ASCII/Unicode U+006A (category Ll: Letter, lowercase)
```

Julia 1.5

The method which accepts a type requires Julia 1.5 or later.

`Base.position` — Function

```
position(s)
```

Get the current position of a stream.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> position(io)
5

julia> skip(io, 10);

julia> position(io)
15

julia> seekend(io);

julia> position(io)
35
```

`Base.seek` — Function

```
seek(s, pos)
```

Seek a stream to the given position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)
```

`Base.seekstart` — Function

```
seekstart(s)
```

Seek a stream to its beginning.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)

julia> seekstart(io);

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
```

`Base.seekend` — Function

```
seekend(s)
```

Seek a stream to its end.

Base.skip — Function

```
skip(s, offset)
```

Seek a stream relative to the current position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> skip(io, 10);

julia> read(io, Char)
'G': ASCII/Unicode U+0047 (category Lu: Letter, uppercase)
```

Base.mark — Function

```
mark(s)
```

Add a mark at the current position of stream `s`. Return the marked position.

See also `unmark`, `reset`, `ismarked`.

Base.unmark — Function

```
unmark(s)
```

Remove a mark from stream `s`. Return `true` if the stream was marked, `false` otherwise.

See also `mark`, `reset`, `ismarked`.

`Base.reset` — Function

```
reset(s)
```

Reset a stream `s` to a previously marked position, and remove the mark. Return the previously marked position. Throw an error if the stream is not marked.

See also `mark`, `unmark`, `ismarked`.

`Base.ismarked` — Function

```
ismarked(s)
```

Return `true` if stream `s` is marked.

See also `mark`, `unmark`, `reset`.

`Base.eof` — Function

```
eof(stream) -> Bool
```

Test whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`. `eof` will return `false` as long as buffered data is still available, even if the remote end of a connection is closed.

`Base.isreadonly` — Function

```
isreadonly(io) -> Bool
```

Determine whether a stream is read-only.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");
julia> isreadonly(io)
true

julia> io = IOBuffer();

julia> isreadonly(io)
false
```

`Base.iswritable` — Function

```
iswritable(io) -> Bool
```

Return `true` if the specified IO object is writable (if that can be determined).

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           iswritable(io)
       end
true

julia> open("myfile.txt", "r") do io
           iswritable(io)
       end
false

julia> rm("myfile.txt")
```

`Base.isreadable` — Function

```
isreadable(io) -> Bool
```

Return `true` if the specified IO object is readable (if that can be determined).

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           isreadable(io)
       end
false

julia> open("myfile.txt", "r") do io
           isreadable(io)
       end
true

julia> rm("myfile.txt")
```

`Base.isopen` — Function

```
isopen(object) -> Bool
```

Determine whether an object - such as a stream or timer – is not yet closed. Once an object is closed, it will never produce a new event. However, since a closed stream may still have data to read in its buffer, use `eof` to check for the ability to read data. Use the `FileWatching` package to be notified when a stream might be writable or readable.

Examples

```
julia> io = open("my_file.txt", "w+");

julia> isopen(io)
true

julia> close(io)

julia> isopen(io)
false
```

`Base.fd` — Function

```
fd(stream)
```

Return the file descriptor backing the stream or file. Note that this function only applies to synchronous `File`'s and `IOStream`'s not to any of the asynchronous streams.

`Base.redirect_stdout` — Function

```
redirect_stdout([stream]) -> (rd, wr)
```

Create a pipe to which all C and Julia level `stdout` output will be redirected. Returns a tuple `(rd, wr)` representing the pipe ends. Data written to `stdout` may now be read from the `rd` end of the pipe. The `wr` end is given for convenience in case the old `stdout` object was cached by the user and needs to be replaced elsewhere.

If called with the optional `stream` argument, then returns `stream` itself.

> **Note**
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.redirect_stdout` — Method

```
redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `stdout` to `stream`. Upon completion, `stdout` is restored to its prior setting.

> **Note**
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.redirect_stderr` — Function

```
redirect_stderr([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `stderr`.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.redirect_stderr` — Method

```
redirect_stderr(f::Function, stream)
```

Run the function `f` while redirecting `stderr` to `stream`. Upon completion, `stderr` is restored to its prior setting.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.redirect_stdin` — Function

```
redirect_stdin([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `stdin`. Note that the order of the return tuple is still `(rd, wr)`, i.e. data to be read from `stdin` may be written to `wr`.

> Note
>
> `stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.redirect_stdin` — Method

```
redirect_stdin(f::Function, stream)
```

Run the function `f` while redirecting `stdin` to `stream`. Upon completion, `stdin` is restored to its prior setting.

Note

`stream` must be a `TTY`, a `Pipe`, or a socket.

`Base.readchomp` — Function

```
readchomp(x)
```

Read the entirety of `x` as a string and remove a single trailing newline if there is one. Equivalent to `chomp(read(x, String))`.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.
       end;

julia> readchomp("my_file.txt")
"JuliaLang is a GitHub organization.\nIt has many members."

julia> rm("my_file.txt");
```

`Base.truncate` — Function

```
truncate(file, n)
```

Resize the file or buffer given by the first argument to exactly `n` bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.")
```

```
35

julia> truncate(io, 15)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=f

julia> String(take!(io))
"JuliaLang is a "

julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.");

julia> truncate(io, 40);

julia> String(take!(io))
"JuliaLang is a GitHub organization.\0\0\0\0\0"
```

`Base.skipchars` — Function

```
skipchars(predicate, io::IO; linecomment=nothing)
```

Advance the stream `io` such that the next-read character will be the first remaining for which `predicate` returns `false`. If the keyword argument `linecomment` is specified, all characters from that character until the start of the next line are ignored.

Examples

```
julia> buf = IOBuffer("    text")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=

julia> skipchars(isspace, buf)
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=

julia> String(readavailable(buf))
"text"
```

`Base.countlines` — Function

```
countlines(io::IO; eol::AbstractChar = '\n')
```

Read `io` until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than `'\n'` are supported by passing them as the second argument. The last non-empty line of `io` is counted even if it does not end with the EOL, matching the length returned by `eachline` and `readlines`.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n");

julia> countlines(io)
1

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io)
1

julia> countlines(io, eol = '.')
0
```

`Base.PipeBuffer` — Function

```
PipeBuffer(data::Vector{UInt8}=UInt8[]; maxsize::Integer = typemax(Int))
```

An `IOBuffer` that allows reading and performs writes by appending. Seeking and truncating are not supported. See `IOBuffer` for the available constructors. If `data` is given, creates a `PipeBuffer` to operate on a data vector, optionally specifying a size beyond which the underlying `Array` may not be grown.

`Base.readavailable` — Function

```
readavailable(stream)
```

Read all available data on the stream, blocking the task only if no data is available. The result is a `Vector{UInt8}`.

`Base.IOContext` — Type

```
IOContext
```

`IOContext` provides a mechanism for passing output configuration settings among `show` methods.

In short, it is an immutable dictionary that is a subclass of `IO`. It supports standard dictionary operations such as `getindex`, and can also be used as an I/O stream.

`Base.IOContext` — Method

```
IOContext(io::IO, KV::Pair...)
```

Create an `IOContext` that wraps a given stream, adding the specified `key=>value` pairs to the properties of that stream (note that `io` can itself be an `IOContext`).

- use `(key => value) in io` to see if this particular combination is in the properties set
- use `get(io, key, default)` to retrieve the most recent value for a particular key

The following properties are in common use:

- `:compact`: Boolean specifying that values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements. `:compact` output should not contain line breaks.
- `:limit`: Boolean specifying that containers should be truncated, e.g. showing … in place of most elements.
- `:displaysize`: A `Tuple{Int,Int}` giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the `displaysize` function.
- `:typeinfo`: a `Type` characterizing the information already printed concerning the type of the object about to be displayed. This is mainly useful when displaying a collection of objects of

the same type, so that redundant type information can be avoided (e.g. `[Float16(0)]` can be shown as "Float16[0.0]" instead of "Float16[Float16(0.0)]" : while displaying the elements of the array, the `:typeinfo` property will be set to `Float16`).

- `:color`: Boolean specifying whether ANSI color/escape codes are supported/expected. By default, this is determined by whether `io` is a compatible terminal and by any `--color` command-line flag when `julia` was launched.

Examples

```
julia> io = IOBuffer();

julia> printstyled(IOContext(io, :color => true), "string", color=:red)

julia> String(take!(io))
"\e[31mstring\e[39m"

julia> printstyled(io, "string", color=:red)

julia> String(take!(io))
"string"
```

```
julia> print(IOContext(stdout, :compact => false), 1.12341234)
1.12341234
julia> print(IOContext(stdout, :compact => true), 1.12341234)
1.12341
```

```
julia> function f(io::IO)
           if get(io, :short, false)
               print(io, "short")
           else
               print(io, "loooooong")
           end
       end
f (generic function with 1 method)

julia> f(stdout)
loooooong
julia> f(IOContext(stdout, :short => true))
short
```

`Base.IOContext` — Method

```
IOContext(io::IO, context::IOContext)
```

Create an `IOContext` that wraps an alternate `IO` but inherits the properties of `context`.

Text I/O

`Base.show` — Method

```
show([io::IO = stdout], x)
```

Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

To customize human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` instead. Checking the `:compact` `IOContext` property of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Examples

```
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

`Base.summary` — Function

```
summary(io::IO, x)
str = summary(x)
```

Print to a stream `io`, or return a string `str`, giving a brief description of a value. By default returns `string(typeof(x))`, e.g. `Int64`.

For arrays, returns a string of size and type info, e.g. `10-element Array{Int64,1}`.

Examples

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Array{Float64,1}"
```

`Base.print` — Function

```
print([io::IO], xs...)
```

Write to `io` (or to the default output stream `stdout` if `io` is not given) a canonical (un-decorated) text representation. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

`print` falls back to calling `show`, so most types should just define `show`. Define `print` if your type has a separate "plain" representation. For example, `show` displays strings with quotes, and `print` displays strings without quotes.

`string` returns the output of `print` as a string.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello", ' ', :World!)

julia> String(take!(io))
"Hello World!"
```

`Base.println` — Function

```
println([io::IO], xs...)
```

Print (using `print`) `xs` followed by a newline. If `io` is not supplied, prints to `stdout`.

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello, world")

julia> String(take!(io))
"Hello, world\n"
```

`Base.printstyled` — Function

```
printstyled([io], xs...; bold::Bool=false, color::Union{Symbol,Int}=:normal)
```

Print `xs` in a color specified as a symbol or integer, optionally in bold.

`color` may take any of the values `:normal`, `:default`, `:bold`, `:black`, `:blink`, `:blue`, `:cyan`, `:green`, `:hidden`, `:light_black`, `:light_blue`, `:light_cyan`, `:light_green`, `:light_magenta`, `:light_red`, `:light_yellow`, `:magenta`, `:nothing`, `:red`, `:reverse`, `:underline`, `:white`, or `:yellow` or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword `bold` is given as `true`, the result will be printed in bold.

`Base.sprint` — Function

```
sprint(f::Function, args...; context=nothing, sizehint=0)
```

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string. `context` can be either an `IOContext` whose properties will be used, or a `Pair` specifying a property and its value. `sizehint` suggests the capacity of the buffer (in bytes).

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `f`. The optional `sizehint` is a suggested size (in bytes) to allocate for the buffer used to write the string.

Examples

```
julia> sprint(show, 66.66666; context=:compact => true)
"66.6667"

julia> sprint(showerror, BoundsError([1], 100))
"BoundsError: attempt to access 1-element Array{Int64,1} at index [100]"
```

`Base.showerror` — Function

```
showerror(io, e)
```

Show a descriptive representation of an exception object `e`. This method is used to display the exception after a call to `throw`.

Examples

```
julia> struct MyException <: Exception
           msg::AbstractString
       end

julia> function Base.showerror(io::IO, err::MyException)
           print(io, "MyException: ")
           print(io, err.msg)
       end

julia> err = MyException("test exception")
MyException("test exception")

julia> sprint(showerror, err)
"MyException: test exception"

julia> throw(MyException("test exception"))
ERROR: MyException: test exception
```

Base.dump — Function

```
dump(x; maxdepth=8)
```

Show every part of the representation of a value. The depth of the output is truncated at `maxdepth`.

Examples

```
julia> struct MyStruct
           x
           y
       end

julia> x = MyStruct(1, (2,3));

julia> dump(x)
MyStruct
  x: Int64 1
  y: Tuple{Int64,Int64}
    1: Int64 2
    2: Int64 3

julia> dump(x; maxdepth = 1)
MyStruct
  x: Int64 1
  y: Tuple{Int64,Int64}
```

Base.Meta.@dump — Macro

```
@dump expr
```

Show every part of the representation of the given expression. Equivalent to `dump(:(expr))`.

Base.readline — Function

```
readline(io::IO=stdin; keep::Bool=false)
readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to `stdin`). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with `'\n'` or `"\r\n"` or the end of an input stream. When `keep` is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When `keep` is true, they are returned as part of the line.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.
       end
57

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
```

`Base.readuntil` — Function

```
readuntil(stream::IO, delim; keep::Bool = false)
readuntil(filename::AbstractString, delim; keep::Bool = false)
```

Read a string from an I/O stream or a file, up to the given delimiter. The delimiter can be a `UInt8`, `AbstractChar`, string, or vector. Keyword argument `keep` controls whether the delimiter is included in the result. The text is assumed to be encoded in UTF-8.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.
       end
57

julia> readuntil("my_file.txt", 'L')
```

```
"Julia"

julia> readuntil("my_file.txt", '.', keep = true)
"JuliaLang is a GitHub organization."

julia> rm("my_file.txt")
```

`Base.readlines` — Function

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.
       end
57

julia> readlines("my_file.txt")
2-element Array{String,1}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Array{String,1}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

`Base.eachline` — Function

```
eachline(io::IO=stdin; keep::Bool=false)
eachline(filename::AbstractString; keep::Bool=false)
```

Create an iterable `EachLine` object that will yield each line from an I/O stream or a file. Iteration calls `readline` on the stream argument repeatedly with `keep` passed through, determining whether trailing end-of-line characters are retained. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the `EachLine` object is garbage collected.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\n It has many members
       end;

julia> for line in eachline("my_file.txt")
           print(line)
       end
JuliaLang is a GitHub organization. It has many members.

julia> rm("my_file.txt");
```

`Base.displaysize` — Function

```
displaysize([io::IO]) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this `IO` object. If no input is provided, the environment variables `LINES` and `COLUMNS` are read. If those are not set, a default size of `(24, 80)` is returned.

Examples

```
julia> withenv("LINES" => 30, "COLUMNS" => 100) do
           displaysize()
       end
(30, 100)
```

To get your TTY size,

```
julia> displaysize(stdout)
(34, 147)
```

Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function `display(x)` to request the richest available multimedia display of a Julia object `x` (with a plain-text fallback).
- Overloading `show` allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.
- Multimedia-capable display backends may be registered by subclassing a generic `AbstractDisplay` type and pushing them onto a stack of display backends via `pushdisplay`.

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

`Base.Multimedia.AbstractDisplay` — Type

```
AbstractDisplay
```

Abstract supertype for rich display output devices. `TextDisplay` is a subtype of this.

`Base.Multimedia.display` — Function

```
display(x)
display(d::AbstractDisplay, x)
display(mime, x)
display(d::AbstractDisplay, mime, x)
```

AbstractDisplay `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text `stdout` output as a fallback. The `display(d, x)` variant attempts to display `x` on the given display `d` only, throwing a `MethodError` if `d` cannot display objects of this type.

In general, you cannot assume that `display` output goes to `stdout` (unlike `print(x)` or `show(x)`). For example, `display(x)` may open up a separate window with an image. `display(x)` means "show `x` in the best way you can for the current output device(s)." If you want REPL-like text output that is guaranteed to go to `stdout`, use `show(stdout, "text/plain", x)` instead.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display `x` using the requested MIME type *only*, throwing a `MethodError` if this type is not supported by either the display(s) or by `x`. With these variants, one can also supply the "raw" data in the requested MIME type by passing `x::AbstractString` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{UInt8}` (for binary MIME types).

To customize how instances of a type are displayed, overload `show` rather than `display`, as explained in the manual section on custom pretty-printing.

`Base.Multimedia.redisplay` — Function

```
redisplay(x)
redisplay(d::AbstractDisplay, x)
redisplay(mime, x)
redisplay(d::AbstractDisplay, mime, x)
```

By default, the `redisplay` functions simply call `display`. However, some display backends may override `redisplay` to modify an existing display of `x` (if any). Using `redisplay` is also a hint to the backend that `x` may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

`Base.Multimedia.displayable` — Function

```
displayable(mime) -> Bool
displayable(d::AbstractDisplay, mime) -> Bool
```

Returns a boolean value indicating whether the given `mime` type (string) is displayable by any of the displays in the current display stack, or specifically by the display `d` in the second variant.

`Base.show` — Method

```
show(io::IO, mime, x)
```

The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O stream `io` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type `T`, it is only necessary to define a new `show` method for `T`, via: `show(io, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `io`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `AbstractDisplay` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the `Day` type uses `1 day` as the output for the `text/plain` MIME type, and `Day(1)` as the output of 2-argument `show`.

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

`Base.Multimedia.showable` — Function

```
showable(mime, x)
```

Returns a boolean value indicating whether or not the object `x` can be written as the given `mime` type.

(By default, this is determined automatically by the existence of the corresponding `show` method for `typeof(x)`. Some types provide custom `showable` methods; for example, if the available MIME formats depend on the *value* of `x`.)

Examples

```
julia> showable(MIME("text/plain"), rand(5))
true

julia> showable("img/png", rand(5))
false
```

`Base.repr` — Method

```
repr(mime, x; context=nothing)
```

Returns an `AbstractString` or `Vector{UInt8}` containing the representation of `x` in the requested `mime` type, as written by `show(io, mime, x)` (throwing a `MethodError` if no appropriate `show` is available). An `AbstractString` is returned for MIME types with textual representations (such as `"text/html"` or `"application/postscript"`), whereas binary data is returned as `Vector{UInt8}`. (The function `istextmime(mime)` returns whether or not Julia treats a given `mime` type as text.)

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

As a special case, if `x` is an `AbstractString` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `repr` function assumes that `x` is already in the requested `mime` format and simply returns `x`. This special case does not apply to the `"text/plain"` MIME type. This is useful so that raw data can be passed to `display(m::MIME, x)`.

In particular, `repr("text/plain", x)` is typically a "pretty-printed" version of `x` designed for human consumption. See also `repr(x)` to instead return a string corresponding to `show(x)` that may be closer to how the value of `x` would be entered in Julia.

Examples

```
julia> A = [1 2; 3 4];

julia> repr("text/plain", A)
"2×2 Array{Int64,2}:\n 1  2\n 3  4"
```

`Base.Multimedia.MIME` — Type

```
MIME
```

A type representing a standard internet data format. "MIME" stands for "Multipurpose Internet Mail Extensions", since the standard was originally used to describe multimedia attachments to email messages.

A `MIME` object can be passed as the second argument to `show` to request output in that format.

Examples

```
julia> show(stdout, MIME("text/plain"), "hi")
"hi"
```

`Base.Multimedia.@MIME_str` — Macro

```
@MIME_str
```

A convenience macro for writing `MIME` types, typically used when adding methods to `show`. For example the syntax `show(io::IO, ::MIME"text/html", x::MyType) = ...` could be used to define how to write an HTML representation of `MyType`.

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling `display(x)` on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype `D` of the abstract class `AbstractDisplay`. Then, for each MIME type (`mime` string) that can be displayed on `D`, one should define a function `display(d::D, ::MIME"mime", x) = ...` that displays `x` as that MIME type, usually by calling `show(io, mime, x)` or `repr(io, mime, x)`. A `MethodError` should be thrown if `x` cannot be displayed as that MIME type; this is automatic if one calls `show` or `repr`. Finally, one should define a function `display(d::D, x)` that queries `showable(mime, x)` for the `mime` types supported by `D` and displays the "best" one; a `MethodError` should be thrown if no supported MIME types are found for `x`. Similarly, some subtypes may wish to override `redisplay(d::D, ...)`. (Again, one should `import Base.display` to add new methods to `display`.) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display "handle" of some type). The display functions for `D` can then be called directly, but they can also be invoked automatically from `display(x)` simply by pushing a new display onto the display-backend stack with:

`Base.Multimedia.pushdisplay` — Function

```
pushdisplay(d::AbstractDisplay)
```

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

`Base.Multimedia.popdisplay` — Function

```
popdisplay()
popdisplay(d::AbstractDisplay)
```

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

`Base.Multimedia.TextDisplay` — Type

```
TextDisplay(io::IO)
```

Returns a `TextDisplay <: AbstractDisplay`, which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

`Base.Multimedia.istextmime` — Function

```
istextmime(m::MIME)
```

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

```
julia> istextmime(MIME("text/plain"))
true

julia> istextmime(MIME("img/png"))
false
```

Network I/O

`Base.bytesavailable` — Function

```
bytesavailable(io)
```

Return the number of bytes available for reading before a read from this stream or buffer will block.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> bytesavailable(io)
34
```

`Base.ntoh` — Function

```
ntoh(x)
```

Convert the endianness of a value from Network byte order (big-endian) to that used by the Host.

`Base.hton` — Function

```
hton(x)
```

Convert the endianness of a value from that used by the Host to Network byte order (big-endian).

`Base.ltoh` — Function

```
ltoh(x)
```

Convert the endianness of a value from Little-endian to that used by the Host.

`Base.htol` — Function

```
htol(x)
```

Convert the endianness of a value from that used by the Host to Little-endian.

`Base.ENDIAN_BOM` — Constant

```
ENDIAN_BOM
```

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value `0x04030201`. Big-endian machines will contain the value `0x01020304`.

« Filesystem Punctuation »