Chapter 9

Getting Started

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

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

```
$ julia

               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.4.1 (2020-04-14)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia> 1 + 2
3

julia> ans
3
```

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

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

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

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

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

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

foo
bar
```

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

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

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

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

See also Scripting for more information on writing Julia scripts.

Julia can be started in parallel mode with either the `-p` or the `--machine-file` options. `-p n` will launch an additional `n` worker processes, while `--machine-file file` will launch a worker for each line in file `file`. The machines defined in `file` must be accessible via a password-less `ssh` login, with Julia installed at the same location as the current host. Each machine definition takes the form `[count*][user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. `count` is the number of workers to spawn on the node, and defaults to 1. The optional `bind-to bind_addr[:port]` specifies the IP address and port that other workers should use to connect to this worker.

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

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

...
```

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

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

Julia 1.1

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

9.1 Resources

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

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