# The great wide world, part 1

# Out-of-tree builds

As opposed to *in-tree builds*, which most Makefiles we've seen to far result in, *out-of-tree builds* place all outputs in a separate directory (or directory tree) from the source files.

Benefits:

- Can build multiple configurations at once from one copy of the source
- Easy to distinguish top-level inputs from outputs and intermediates
- `.gitignore` can be much smaller and less error-prone

# Reproducible builds

*Reproducible builds* (a.k.a. *hermetic builds*) are achieved when a build process guarantees identical output from identical source code, regardless of where or when you run the build. Builds often implicitly depend on attributes like

- The specific installed versions of the compiler, linker, and other tools
- The time the build runs (often included in artifacts for debugging purposes)
- The implementation of system tools like the shell and POSIX utilities

All these, if not explicitly recorded, make the build hard to reproduce. Some build systems make reproducible builds easier by eliminating implicit dependencies on the system: those that can't be removed outright are at least made explicit.

Reproducible builds improve security by allowing anyone to verify that a release of a project was built from the source code it claims to have been.

# Limitations of Make

- Maximum one output per rule
  - Though GNU make has a nonstandard notion of grouped targets
- Out-of-tree builds are opt-in rather than opt-out
- Doesn't facilitate reproducible builds
- Always uses mtime to compare
  - Switching between branches might change mtime and lead to longer builds
  - mtime is at best a heuristic

# Types of build system

Language-agnostic

- Recipes usually call arbitrary tools, as with Make
- Lesser-used: usually Make is "good enough" or a tailor-made system exists

Language-specific

- Support a single language
- Tie into their language's package ecosystem
- Used by most open-source projects in their language

Multi-language

- Bring benefits of language-specific systems to multi-language projects
- Often created by larger companies
- Don't usually integrate with language package ecosystems

# Not every build system is a *build runner*

Nearly all build systems have a concept, either user-visible or internal, of rules with inputs and outputs that are rebuilt only when necessary. But, surprisingly, many build systems don't actually include logic to *run* those rules.

Such systems instead translate their rule definitions into a set of lower-level definitions in the language of a different build system, which then can be invoked to run the build. Common build runners include:

- Ninja (see next slide)
- Make
- MSBuild (bundled with Visual Studio)

# Sampling: Language-agnostic

# Make

For reference, here's an idiomatic Makefile for the project all the following samples will also build.

```
$ make
$ make clean # clean
```

*Makefile*

```
EXE := main

COMMON_FLAGS := -O2

CFLAGS := $(COMMON_FLAGS) -Wall -Wextra -std=gnu99
LDFLAGS := $(COMMON_FLAGS)

$(EXE): main.o log.o

main.o: main.c log.h
log.o: log.c log.h

.PHONY: clean
clean:
    rm -f $(EXE) *.o
```

# Ninja: the ultimate build runner

Language designed to be emitted from a higher-level build system, not to be hand-written.

Focuses on speed and sane defaults (e.g. builds on all cores by default).

Has first-class notion of a "generator", used to regenerate Ninja rules.

Influenced by Tup!

```
$ ninja
$ ninja -t clean # clean
```

*build.ninja*

```
common_flags = -O2
cflags = $common_flags -Wall -Wextra -std=gnu99
ldflags = $common_flags

rule compile
  command = gcc $cflags -c -o $out $in

rule link
  command = gcc $ldflags -o $out $in

build main.o: compile main.c | log.h
build log.o: compile log.c | log.h

build main: link main.o log.o
```

# Rake

Build rules defined using Ruby.

Special functions for defining rules available, but all normal Ruby syntax also available.

Inputs and outputs can be files, but they can also be named rules that are totally internal to the build system.

```
$ rake
$ rake clean # clean
```

*Rakefile*

```
common_flags = "-O2"
$cflags = "#{common_flags} -Wall -Wextra -std=gnu99"
$ldflags = common_flags

def compile_file(c_file, o_file)
  sh "gcc #{$cflags} -c -o #{c_file} #{o_file}"
end

file "main.o": ["main.c", "log.h"] do |t|
  compile_file(t.name, t.prerequisites.first)
end

file "log.o": ["log.c", "log.h"] do |t|
  compile_file(t.name, t.prerequisites.first)
end

file "main" => ["main.o", "log.o"] do |t|
  sh "gcc #{$ldflags} -o #{t.name} #{t.prerequisites.join(' ')}"
end

task :clean do |t|
  sh "rm -f main *.o"
end

task default: ["main"]
```

# Tup

Rules are specified as shell commands, like Make.

No need to specify header dependencies! Hooks into the operating system to monitor what files are read during each command invocation.

Keeps DAG state on disk.
paper link

```
$ tup init
$ mkdir build
$ touch build/tup.config
$ tup
$ rm -r build # clean
```

*Tupfile*

```
COMMON_FLAGS = -O2
CFLAGS = $(COMMON_FLAGS) -Wall -Wextra -std=gnu99
LDFLAGS = $(COMMON_FLAGS)

: foreach main.c log.c |> gcc $(CFLAGS) -c -o %o %f |> %B.o
: main.o log.o |> gcc $(LDFLAGS) -o %o %f |> main
```