GNU Make Build automation

START HERE

`make`
Build the default target

`make -f <file>`
Use another makefile

`make <target>`
Build one target

`make -C <dir>`
Run in another directory

`make -n`
Dry run: show recipes only

`make <VAR>=<value>`
Override a variable

`make -j<N>`
Build N jobs in parallel

`make -B`
Rebuild everything

Makefile rules + variables → `parse` → **Goal target** default: first rule → `compare mtimes` → **Prerequisites** built first, recursively → `if stale` → **Recipe** shell lines run

Make walks the dependency graph and runs only stale recipes

CONCEPTS

A rule says: target is rebuilt when any prerequisite is newer than it. Make compares file timestamps, nothing else. Recipe lines must start with a Tab.

Each recipe line runs in its own shell: cd or exported variables do not carry over to the next line. Join lines with && and a trailing backslash, or set .ONESHELL.

RUNNING MAKE

`make`
Build the first (default) target

`make <target>`
Build a specific target

`make -n`
Print recipes without running them

`make -j<N>`
Run up to N recipes in parallel
-j with no number means unlimited jobs.

`make -f <file>`
Read this file instead of Makefile

`make -C <dir>`
Change to dir before reading makefiles

`make <VAR>=<value>`
Set a variable from the command line
Command-line values beat assignments in the makefile unless it uses override.

`make -B`
Rebuild all targets unconditionally

`make -k`
Keep going after errors

`make -s`
Silent: do not echo recipes

`make -O`
Group output of parallel jobs

RULES

`<target>: <prereqs>`
Rule header; recipe lines follow, each after a Tab

`<a> <b>: <prereqs>`
One rule for several targets

`<target>: | <dir>`
Order-only prerequisite: must exist, no rebuild trigger
Typical use: create an output directory without rebuilding on its mtime.

RULES (cont.)

`.PHONY: <target>`
Target is an action, not a file

`.DEFAULT_GOAL := <target>`
Choose the default target

`<t1> <t2> &: <prereqs>`
Grouped targets: one recipe makes all
Needs GNU Make 4.3 or newer.

`.DELETE_ON_ERROR:`
Remove target if its recipe fails

`.ONESHELL:`
Run a whole recipe in one shell

`include <file>`
Include another makefile; error if missing

`-include <file>`
Include if it exists, no error otherwise

RECIPE LINES

`@<command>`
Do not echo this line

`-<command>`
Ignore a failure of this line

`$(MAKE) -C <dir>`
Recurse into a subdirectory
Use $(MAKE), not make, so flags and the jobserver are passed down.

`$$<var>`
Literal $ for the shell
Make expands $ first, so shell variables need $$.

`\ (end of line)`
Continue the line

`export <VAR>`
Pass variable to recipes' environment

VARIABLES

`<VAR> = <value>`
Recursive: expanded each time it is used

`<VAR> := <value>`
Simple: expanded once, at assignment

`<VAR> ?= <value>`
Set only if not already set

`<VAR> += <value>`
Append

`$(<VAR>) / ${<VAR>}`
Use a variable

`$(<VAR>:.c=.o)`
Replace suffix in each word

AUTOMATIC VARIABLES

`$@`
The target

AUTOMATIC VARIABLES (cont.)

`$<`
The first prerequisite

`$^`
All prerequisites, no duplicates

`$?`
Prerequisites newer than the target

`$*`
The stem matched by % in a pattern rule

`$(@D) / $(@F)`
Directory and file part of $@

PATTERNS AND BUILT-INS

`%.o: %.c`
Pattern rule: any .o from the matching .c

`$(CC)`
C compiler (default cc)

`$(CFLAGS) / $(CPPFLAGS)`
Compile and preprocessor flags

`$(LDFLAGS) / $(LDLIBS)`
Link flags and libraries

`$(CXX) / $(CXXFLAGS)`
C++ compiler and flags

`$(RM)`
Built-in rm -f

FUNCTIONS

`$(wildcard <glob>)`
Files matching a glob

`$(patsubst <pat>,<repl>,<text>)`
Pattern replace in each word

`$(subst <from>,<to>,<text>)`
Plain text replace

`$(shell <command>)`
Output of a shell command

`$(foreach <v>,<list>,<text>)`
Expand text per word

`$(filter <pat>,<text>)`
Keep words matching

`$(filter-out <pat>,<text>)`
Drop words matching

`$(dir <names>) / $(notdir <names>)`
Directory part / file part

`$(addprefix <p>,<names>) / $(addsuffix <s>,<names>)`
Add prefix or suffix to each word

`$(error <text>) / $(warning <text>)`
Abort or warn at parse time

FUNCTIONS (cont.)

`$(info <text>)`
Print a message while parsing

CONDITIONALS

`ifeq (<a>,<b>) ... else ... endif`
Text equality test

`ifneq (<a>,<b>)`
Text inequality test

`ifdef <VAR>`
True if variable has a non-empty value

`ifndef <VAR>`
True if variable is empty or unset

DEBUGGING

`make -d`
Verbose debug output

`make --debug=b`
Basic debugging: why targets rebuild

`make -p`
Print rules and variables database

`make --trace`
Show why each recipe runs
Added in GNU Make 4.0.

`make --warn-undefined-variables`
Warn on unset variable use

RECIPES

Self-documenting help target

```
.PHONY: help
help:
@grep -E '^[a-z-]+:.*##'
$(MAKEFILE_LIST) \
  | sed 's/:.*## /\t/'

build: ## compile the project
```

Compile C into a build directory

```
SRC := $(wildcard src/*.c)
OBJ := $(SRC:src/%.c=build/%.o)

app: $(OBJ)
$(CC) -o $@ $^ $(LDLIBS)

build/%.o: src/%.c | build
$(CC) $(CPPFLAGS) $(CFLAGS) -c
-o $@ $<

build:
mkdir -p $@
```

Auto-generated header dependencies

```
CPPFLAGS += -MMD -MP
-include $(OBJ:.o=.d)
```

Run a sub-make per directory

```
SUBDIRS := lib app
.PHONY: all $(SUBDIRS)
all: $(SUBDIRS)
$(SUBDIRS):
$(MAKE) -C $@
```