

Linux Systems and Open Source Software

GNU Make

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2022



Outline

- Build automation
- GNU Make



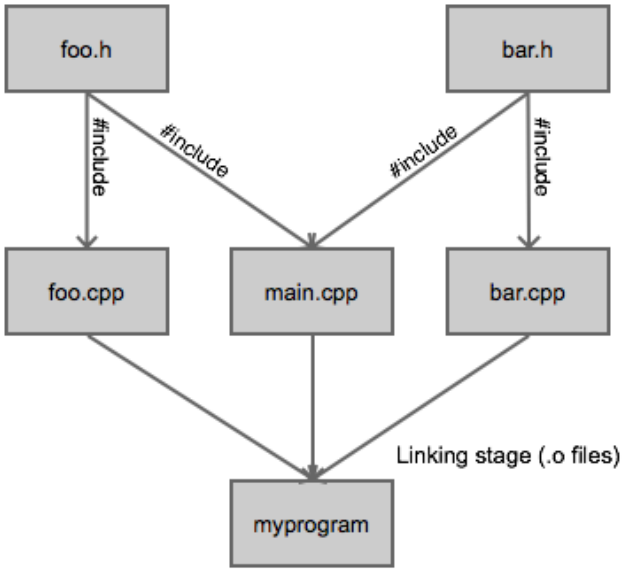
BUILD AUTOMATION



Build a Real Project

- Complex dependencies often exist in a SW project
  - It is important to use the correct ***order*** to build the project files
- The commands to build the project depend on the status of file(s)
  - E.g., when **foo.h** is modified, both **foo.cpp** and **main.cpp** need to be recompiled, and the binary **myprogram** needs to be linked
  - When **foo.cpp** is modified, it needs to be recompiled, and the binary needs to be linked
- Project building is an important task in the iterative SW development process
  - Simply execute all of the commands to build the project is possible, but costs time
  - Sometimes, it takes hours to build a big project

File dependency




```
# Commands to build the project.
$ g++ -Wall -O2 -c foo.cpp
$ g++ -Wall -O2 -c bar.cpp
$ g++ -Wall -O2 -c main.cpp
$ g++ -Wall -O2 -o myprogram main.o foo.o bar.o
```

Courtesy of http://csci221.artifice.cc/lecture/makefiles.html



Build Automation (組建自動化)

- The process of automating the creation of a software build and the associated processes
  - including **compiling computer source code into binary code, packaging binary code, and running automated tests**
- The level of automation
  - Makefile - level
    - **Make-based tools**, e.g., GNU Make
    - Non-Make-based tools
  - Build script (or Makefile) generation tools, e.g., cmake, catkin_make
  - Continuous-integration tools
  - Configuration-management tools
  - Meta-build tools or **package managers**, e.g., maven avoiding manual installs and updates

Courtesy of https://en.wikipedia.org/wiki/Build_automation



Basics of a Rule

Variables

Recursive Use of Make

Functions in Make

Misc. (Echoing and Overriding Variables)

GNU MAKE



The GNU Project

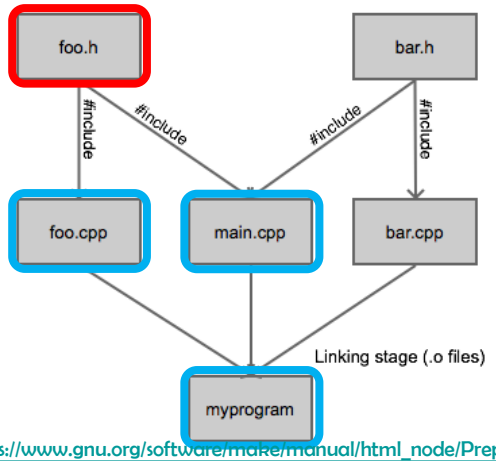
- The GNU Project is a **free software, mass collaboration project**
  - that Richard Stallman announced on September 27, 1983
  - Its goal is to give computer users freedom and control in their use of their computers and computing devices by collaboratively developing and publishing software
  - **that gives everyone the rights to freely run the software, copy and distribute it, study it, and modify it**
- The founding goal of the project was to build **a free operating system**,
  - and if possible, "everything useful that normally comes with a Unix system so that one could get along without any software that is not free"
  - Stallman decided to call this operating system GNU (a recursive acronym meaning "GNU's not Unix!"), basing its design on that of Unix, a proprietary operating system
- Combing with Linux kernel, the software utilities developed by the GNU Project establish **the first free operating system**, commonly known as **Linux**
  - Software from the project may include an editor (GNU Emacs), a source level debugger, a yacc-compatible parser generator, a linker, an assembler, a portable optimizing C compiler (GCC), and utilities, such as Bash, ls, grep, awk, make and ld

Courtesy of https://en.wikipedia.org/wiki/GNU_Project



GNU Make

- **Make** utility automatically determines
  - which pieces of a large program need to be recompiled, and
  - issues commands to recompile them
- **Makefile data base** and **the last-modification times of the files** are used to decide which of the files need to be updated
  - For each of those to-be-updated files, **make** issues the recipes recorded in the data base
  - The example below shows the build commands used when **foo.h** is updated



Only files related to **foo.h** are recompiled

```
/tmp/test
$ make
g++ -Wall -O2 -c main.cpp
g++ -Wall -O2 -c foo.cpp
g++ -Wall -O2 -c bar.cpp
g++ -Wall -O2 -o myprogram main.o foo.o bar.o

/tmp/test
$ touch foo.h

/tmp/test
$ make
g++ -Wall -O2 -c main.cpp
g++ -Wall -O2 -c foo.cpp
g++ -Wall -O2 -o myprogram main.o foo.o bar.o
```



Makefile

- Before the **make** process starts, *makefile* should be in place,
  - describing the **relationships** among files in your project and
  - providing commands to handle each file
- A *makefile* is often formed by ***rules***,
  - each of which consists of ***target***, ***prerequisites***, and ***recipe***
  - A **rule** explains how and when to re-build certain files, which are described in the *target* of the particular rule
  - **Make** carries out the recipe on the prerequisites to create or update the target

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

Must be a "**tab**" not "**spaces**"

Courtesy of https://www.gnu.org/software/make/manual/html_node/Preparing.html
https://www.gnu.org/software/make/manual/html_node/Rule-Syntax.html#index-_0024_002c-in-rules



```
target ... : prerequisites ...
        recipe
        ...
```

Basis of a Rule

Dependency

- A **target** is usually **a file name** that is generated by a program
  - A target can also be the name of **an action to carry out**, such as '**clean**'
  - Usually, there is only one target per rule, but there is a reason to have more
- A list of file names separated by spaces are put in the ***prerequisites***
  - A ***target*** is out of date if it does not exist or if it is older than any of the prerequisites by comparison of last-modification times
  - The contents of the *target* file are computed based on information in the *prerequisites*, so if any of the *prerequisites* changes, the contents of the existing *target* file are no longer necessarily valid
- A **recipe** contains one or more lines of commands carrying out by the shell
  - A *recipe* line start with a tab character, or may appear on the same line, with a semicolon

Courtesy of https://www.gnu.org/software/make/manual/html_node/Rule-Syntax.html#index-_0024_002c-in-rules

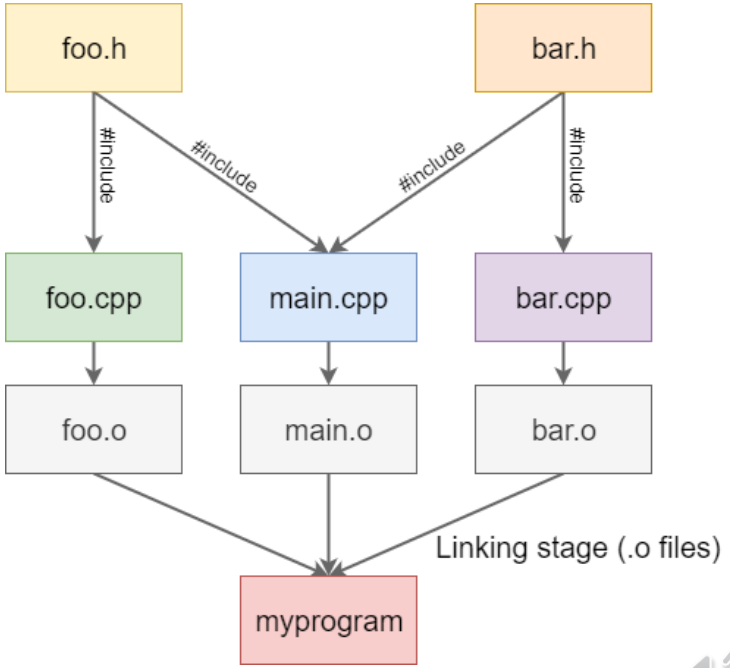


A Makefile Example

- Type **make all** to build the program

```
all: myprogram
.PHONY: all
foo.o: foo.cpp foo.h
        g++ -Wall -O2 -c foo.cpp

bar.o: bar.cpp bar.h
        g++ -Wall -O2 -c bar.cpp

main.o: main.cpp foo.h bar.h
        g++ -Wall -O2 -c main.cpp

myprogram: main.o foo.o bar.o
        g++ -Wall -O2 -o myprogram main.o foo.o bar.o
```

Dependency





More on the *Targets*

- A **target** is usually **a file name** that is generated by a program
- **A target can also be the name of an action to carry out** when you make an explicit request
- To avoid the conflict between **the file name** and **action name**, and to improve performance
  - A **phony** target is used for marking the special target, which is used by making an explicit request
  - *Typical phony targets: **all, clean, install**

```
.PHONY: clean
clean:
        rm -f *.o
```

'**clean**' is not a filename, but an action name

Courtesy of https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html#Phony-Targets
https://www.gnu.org/software/make/manual/html_node/Standard-Targets.html



More on the *Targets* (Cont.)

- It is common to create the **clean** target to
  - delete all files in the current or other directory that are created during the process of building the program
- An issue will be raised when **clean** is one of the file names
  - **make** treats **clean** as the file name, but we *think* **clean** is action name
  - It can be resolved by declaring the **PHONY** target for **clean**

```
all: file2
file2: file1
        cp file1 file2
file1:
        touch file1
        touch clean
clean:
        rm -f file1 file2 clean
```

```
.PHONY: all clean
all: file2
file2: file1
        cp file1 file2
file1:
        touch file1
        touch clean
clean:
        rm -f file1 file2 clean
```

There is a file named ***clean***

'**make clean**' does not work as expected

```
$ ls
clean  file1  file2  Makefile
$ make clean
make: 'clean' is up to date.
```

```
$ ls
clean  file1  file2  Makefile
$ make clean
rm -f file1 file2 clean
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html#Phony-Targets



More on the *Targets* (Cont.)

- The **order of rules** is not significant
  - except for determining the default goal (**.DEFAULT_GOAL**) the target for **make** to consider, if you do not otherwise specify one
  - The default goal is the **target** of the **first rule** in the *makefile*
- It is a common practice that the first rule is the one for
  - **compiling the entire program** or
  - **all the programs described by the *makefile***
  - Its target name is often called '**all**'

A makefile to build three programs

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o
prog2 : prog2.o
        cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html#Phony-Targets



Variables

- A ***variable*** is a name defined in a makefile to represent a string of text, called the variable's *value*
  - Perform value reference with a dollar sign '**$**' and parentheses (or braces), e.g., '**$(foo)**' or '**${foo}**' is a valid reference to **foo**
  - Variable references can be used in any context
  - E.g., targets, prerequisites, recipes, most directives, and new variable values

A makefile to build C++ programs

```
CXX := g++
CPPFLAGS := -Wall -O2
objs := main.o foo.o bar.o

foo.o: foo.cpp foo.h
        $(CXX) $(CPPFLAGS) -c foo.cpp
myprogram: $(objs)
        $(CXX) $(CPPFLAGS) -o myprogram $(objs)
```

https://www.gnu.org/software/make/manual/html_node/Reference.html#Reference

Flavors of Variables



- There are **two *flavors*** for value assignment
  - They are distinguished in **how they are defined** and in **what they do when expanded**
- ***Recursively*** expanded variables use '**=**'
  - The value you specify is installed verbatim;
  - that is, if it contains references to other variables, these references are ***expanded*** whenever this variable is substituted
  - The example"**echo $(foo)**" expands to '**$(bar)**' which expands to '$(ugh)' which finally expands to "Huh?"

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
    echo $(foo)
```

- ***Simply*** expanded variables use '**:=**' or '**::=**'
  - The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined

```
x := foo
y := $(x) bar
x := later
```

=

```
y := foo bar
x := later
```

- Others
  - A conditional variable assignment operator, '**?=**'; it only has an effect if the variable is not yet defined
  - '**+=**' appends more text to the value of a pre-defined variable

```
FOO ?= bar
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Flavors.html#Flavors
https://www.gnu.org/software/make/manual/html_node/Appending.html



Automatic Variables

- Imagine that …
  - there are many **.c** files to be compiled into the corresponding **.o** files, but we do not want to type the file names manually in the makefile
  - ***Automatic variables*** and ***pattern rules*** are there to solve the problem

- Automatic variables have values computed afresh for each rule while it is executed,
  - based on the target and prerequisites of the rule

Examples of automatic variables:

- **$@**: The file name of the rule target
- **$<**: The name of the first prerequisite
- **$^**: All the prerequisites, with spaces between them

```
objs := main.o foo.o bar.o
foo.o: foo.cpp foo.h
        $(CXX) $(CPPFLAGS) -c $<
myprogram: $(objs)
        $(CXX) $(CPPFLAGS) -o $@ $^
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html



Pattern Matching in Pattern Rules

- A **pattern rule** looks like an ordinary rule, except that its **target** contains the character '**%**'
  - The target is considered **a pattern for matching file names**
  - the '**%**' can match any nonempty substring called ***stem***, while other characters match only themselves

```
%.o : %.c
        recipe...
```

- A pattern rule '**%.o : %.c**' says how to make any file **stem.o** from another file **stem.c**
- If someone needs '**foo.o**', prerequisite is **foo.c**

  - E.g., if someone needs **foo.o**, prerequisites are **foo.cpp** and **foo.h**

```
%.o: %.cpp %.h
        $(CXX) $(CPPFLAGS) -c $<
```

  - Note that expansion using '**%**' in pattern rules occurs after any variable or function expansions, which take place when the makefile is read

Courtesy of https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html#Pattern-Rules



Static Pattern Rules

- Rules specify **multiple targets** and construct the prerequisite names for each target based on the target name
  - They are more general than ordinary rules with multiple targets because **the targets do not have to have identical prerequisites**
  - Their prerequisites must be *analogous*, but not necessarily *identical*

```
targets: target-pattern: prereq-patterns ...
        recipe
```

- The ***targets*** list specifies the targets that the rule applies to; a ***target*** should match ***target-pattern***
- The ***target-pattern*** and ***prereq-patterns*** say how to compute the prerequisites of each target

```
main.o: %.o: %.cpp foo.h bar.h
        $(CXX) $(CPPFLAGS) -c $<
```

Match **main.o** with **%.o**, so the stem is '**main**', and the prerequisites become **main.cpp foo.h bar.h**

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)) : %.o : %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)) : %.elc : %.el
        emacs -f batch-byte-compile $<
```

- **$(filter %.o,$(files))'** is **bar.o** and **lose.o**
- **$(filter %.elc,$(files))'** is **foo.elc**; hence, **$<** is **foo.el**

https://www.gnu.org/software/make/manual/html_node/Pattern-Match.html#Pattern-Match
https://www.gnu.org/software/make/manual/html_node/Static-Usage.html#Static-Usage

Example of Pattern Rules



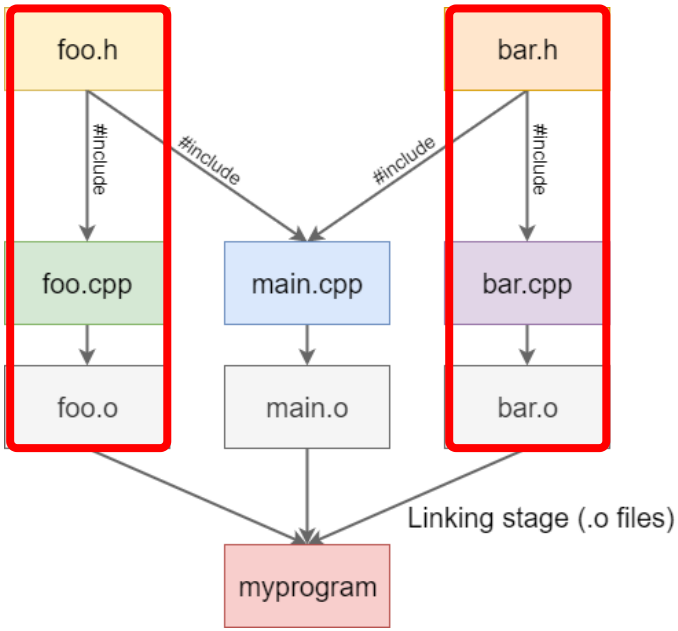
Original

```
foo.o: foo.cpp foo.h
        g++ -Wall -O2 -c foo.cpp
bar.o: bar.cpp bar.h
        g++ -Wall -O2 -c bar.cpp
main.o: main.cpp foo.h bar.h
        g++ -Wall -O2 -c main.cpp
myprogram: main.o foo.o bar.o
        g++ -Wall -O2 -o myprogram main.o foo.o bar.o
```

Revision

```
CXX := g++
CPPFLAGS := -Wall -O2
objs := main.o foo.o bar.o
target := myprogram
.PHONY: all clean
all: $(target)
%.o: %.cpp %.h
        $(CXX) $(CPPFLAGS) -c $<
main.o: %.o: %.cpp foo.h bar.h
        $(CXX) $(CPPFLAGS) -c $<
$(target): $(objs)
        $(CXX) $(CPPFLAGS) -o $@ $^
clean:
        rm -f *.o $(target)
```

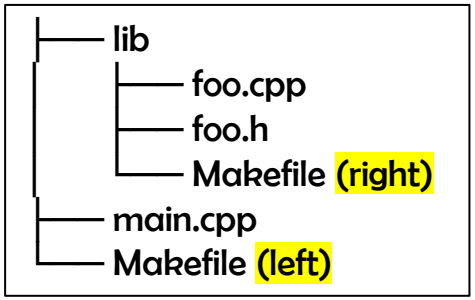
- Extendable
- Flexible
- Readable





Recursive Use of Make

- Use **make** as a command in a makefile
  - It is useful when you want to **separate makefiles for various subsystems** that compose a larger system
  - For example, suppose you have a sub-directory **lib** which has its own makefile, and you would like the containing directory's makefile to run **make** on the sub-directory



- To do it, **make** system should change to directory **lib** before running the makefile
  - with the flag: '**-C dir**' or '**--directory=dir**'
  - The example shows two ways to do it
  - **$(Make)** should be used, instead of **make**

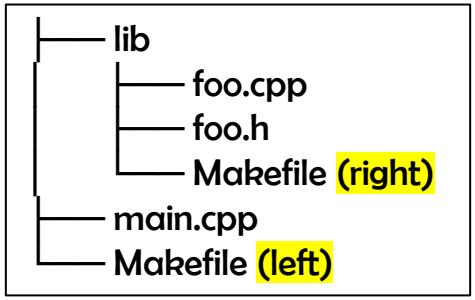
```
# One way to do it
sub-dir:
	cd lib && $(MAKE)

# Another way to do it
sub-dir:
	$(MAKE) -C lib
all: $(target)
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Recursion.html#Recursion



Recursive Use of Make (Cont.)

- The example builds **myprogram**
  - It requires to build the **foo.o** in sub-directory **lib**
  - It also requires to build **main.o**



./Makefile

```
CXX := g++
subdir := ./lib
CPPFLAGS := -Wall -O2 -I$(subdir)
objs := main.o $(subdir)/foo.o
target := myprogram
all: $(target)
$(subdir)/foo.o: %.o: %.cpp %.h
        $(MAKE) -C $(subdir)
main.o: %.o: %.cpp $(subdir)/foo.h
        $(CXX) $(CPPFLAGS) -o $@ -c $<
$(target): $(objs)
        $(CXX) $(CPPFLAGS) -o $@ $^
clean:
        rm -f *.o $(target)
        $(MAKE) -C $(subdir) clean
```

./lib/Makefile

```
CXX := g++
CPPFLAGS := -Wall -O2

target := foo.o
all: $(target)

$(target): %.o: %.cpp %.h
    $(CXX) $(CPPFLAGS) -o $@ -c $<

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

Courtesy of https://www.gnu.org/software/make/manual/html_node/Recursion.html#Recursion



Using Wildcard Characters in File Names

- A single string can specify many files with wildcard characters
  - '*****', '**?**' and '**[...]**' are the wildcard characters in **make**
  - For example, ***.c** specifies a list of all the files (in the working directory) whose names end in '**.c**'
  - You can find the list of supported functions here
- Wildcards can be used
  - in the **recipe** of a rule, where they are expanded by the shell
  - in the **prerequisites** of a rule
  - in the **value assignment** of a variable by using the *wildcard function*

✓ objects := $(wildcard *.o)
✗ objects = *.o

Courtesy of https://www.gnu.org/software/make/manual/html_node/Wildcard-Examples.html#Wildcard-Examples
https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function
http://www.i2symbol.com/symbols/check



wildcard Function

- Wildcard expansion happens **automatically in rules**
  - It does **not** normally take place when a **variable is set**, or
  - inside the **arguments of a function**
  - If you want to do wildcard expansion in such places, you need to use the **wildcard** function
- **$(wildcard pattern...)**
  - This string, used anywhere in a makefile, is replaced by **a space-separated list of existing file names** that match one of the given file name patterns
  - If no existing file name matches a pattern, then that pattern is omitted from the output of the wildcard function

```
SRC := $(wildcard *.c)
```

To get a list of all the C source files in a directory

```
objects := $(patsubst
%.c,%.o,$(wildcard *.c))

foo : $(objects)
      cc -o foo $(objects)
```

- An extension of the above example
- Change the list of C source files into a list of object files by replacing the '**.c**' suffix with '**.o**' in the result
- **objects** now contains the '**.o**' files for the C source files

Courtesy of https://www.gnu.org/software/make/manual/html_node/Wildcard-Examples.html#Wildcard-Examples
https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function



shell Function

- **It communicates with the world outside of make**
  - Perform the same function that backquotes ‘`’ perform in most shells → does *command expansion*
  - Is unlike any other function other than the wildcard function
  - It takes a shell command as part of an argument and evaluates the output of the command

```
# A list of filenames returned by “ls”
files := $(shell ls)

# The content in the file, foo, returned by “cat”
contents := $(shell cat foo)

# A list of filenames within the “src” folder with the suffix with “.c”
CFILES = $(shell find src/ -type f -name '*.c')
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Shell-Function.html#Shell-Function



Suppress Echoing and Ignore Errors

- By default, **make** prints each line of the recipe before it is executed, which is called ***echoing***
- When a line starts with '**@**', the echoing of that line is suppressed

```
hello:
	echo "hello world"
```

```
$ make hello
echo hello world
hello world
```

```
hello:
	@echo "hello world"
```

```
$ make hello
hello world
```

- If there is an error (i.e., the exit status is nonzero), **make** gives up on the current rule and exits
  - To ignore errors, write a '**-**' at the beginning of the line

```
t1:
	mkdir tmp # -mkdir tmp
t2: t1
	touch tmp/foo.txt
```

```
$ make t2
mkdir tmp
mkdir: 無法建立目錄'tmp': File exists
make: *** [Makefile:18: t1] Error 1
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Echoing.html#Echoing
https://www.gnu.org/software/make/manual/html_node/Errors.html#Errors



Arguments Passing (Overriding Variables)

- A command line argument to **make** that contains '=' specifies the value of a variable: "**v=x**"
  - All ordinary assignments of the same variable in the *makefile* are **ignored**
  - It is said they have been *overridden* by the command line argument

```
CXX := g++
CPPFLAGS := -Wall -O2
target := myprogram
INPUT := Alice
all: $(target)
$(target): main.cpp
        $(CXX) $(CPPFLAGS) -o $@ $<
test: $(target)
        ./$(target) $(INPUT)
clean:
        rm -f $(target)
```

```
$ make test
./myprogram Alice
Hello, Alice.
```

```
$ make test INPUT=Bob
./myprogram Bob
Hello, Bob.
```

```
$ make test INPUT='Paul Smith'
./myprogram Paul Smith
Hello, Paul Smith.
```

Courtesy of https://www.gnu.org/software/make/manual/html_node/Overriding.html#Overriding



References

- GNU Make: A Program for Directing Recompilation - Richard M. Stallman, Roland McGrath, Paul D. Smith Version 4.2 May 2016
- GNU Autotools
- What's Apache Ant, Apache Maven, What's the Difference?