



Make

AUTOMATING SOFTWARE BUILDING

Contents with examples.

1. Introduction
2. Preparing
3. Rules
4. Syntax
5. More Resource



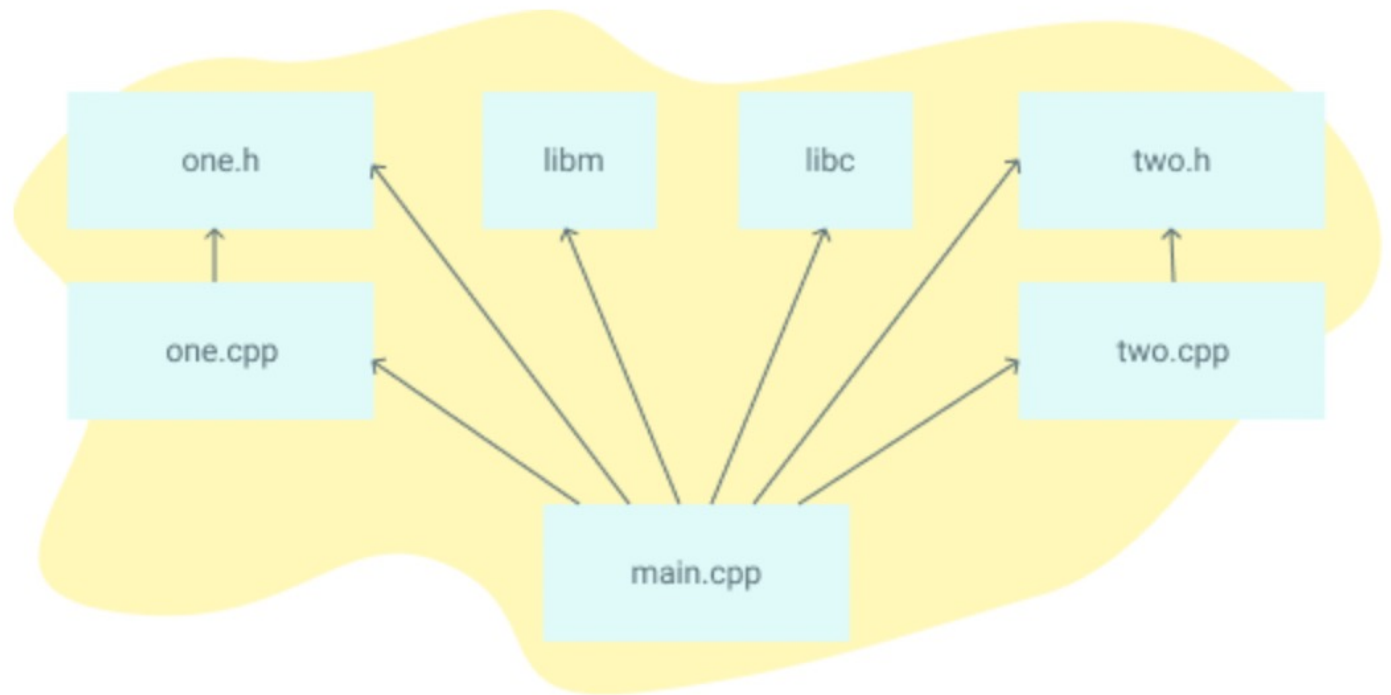
Introduction: Why 'Make'?

If you are developing a project without a powerful IDE...

How to build/update?

Too tedious to explicitly call the compiler each time.

We need a tool to clarify the dependencies and building rules for targets.

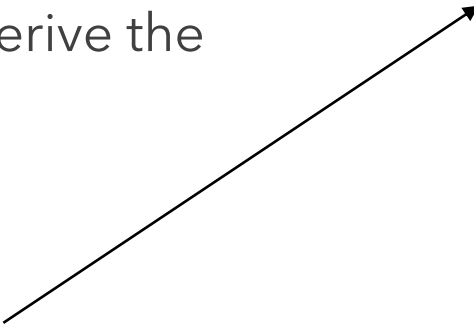


Introduction: What is Make?

Make is a build automation tool that builds executable programs and libraries from source code by reading Makefile which specify how to derive the target program.

Widely used in Unix-like OS.

Simply, 'make' itself is a shell command.



```
~/MakefileDemo$ l
main.c Makefile
~/MakefileDemo$ make
gcc main.c -o main
~/MakefileDemo$ l
main* main.c Makefile
~/MakefileDemo$ ./main
Hello world!
```

```
~$ file /bin/make
/bin/make: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=49151f9534a4154c675f61c96a12fea5cd2552bc,
stripped
~$
```

Introduction: Check your version

Most Linux system would have it pre-installed.

`make --version` to check version number & description.

`make --help` or `man make` to look up arguments.

```
~$ make --version
GNU Make 4.2.1
Built for aarch64-unknown-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
~$ make --help
Usage: make [options] [target] ...
Options:
  -b, -m                        Ignored for compatibility.
  -B, --always-make             Unconditionally make all targets.
  -C DIRECTORY, --directory=DIRECTORY
```

Introduction

What: Not a compiler! But it could let compilers (or other tools) finish specified tasks.

Why: Build & Update project.

Where: Unix-like Systems (Linux), MacOS...

How: Write a file called "Makefile", then `make`.

Preparation: What do we need?

To let `make` know what and how do we build, we need a script file.

Typically, it is names as 'Makefile', or 'makefile' (no extensions).

The script file is written in a specified Makefile language, but it would be easier if you are familiar with shell commands.

Once you execute `make`, it will find this script, otherwise report a vital error message.

```
~/MakefileDemo$ mv Makefile ..  
~/MakefileDemo$ make  
make: *** No targets specified and no makefile found.  Stop.  
~/MakefileDemo$
```


Preparation: Alternative 'Makefile'

However, if we really need to use other names, use '-f'.

``make -f [filename]``

``make`` will treat `[filename]` as 'Makefile'.

Useful when you want to preserve different versions.

```
~/MakefileDemo$ cp Makefile somefile
~/MakefileDemo$ make -f somefile
gcc main.c -o main
~/MakefileDemo$ make
make: 'main' is up to date.
```


Rules: Basic

Usually, Makefile consists of a set of rules. These rules determine what are the target files and how to make them.

A rule generally looks like this:

```
targets: prerequisites  
  command  
  command  
  command
```

There is an indentation before 'command'!

Rules: Basic

```
targets: prerequisites  
command  
command  
command
```

targets: The target file we need, or some intermediate files, or some actions.

Actions could also be called as 'phony' targets, or label.

prerequisites: Things needed before making targets.

For example, we need 'main.c' to make 'main'. 'main.c' is the prerequisites.

command: The commands to create targets, or other shell commands...

The command does not necessarily create the target, if the target is not the final target or the prerequisites of other targets.

Rules: targets

```
1 main: main.c
2      gcc main.c -o main
3 main_1: main.c
4      gcc main.c -o main_1
```

Specifying a target tells make there is a rule for a task.

Not all targets will be made!

By default, only the first target in Makefile will be made.

```
~/MakefileDemo$ make
gcc main.c -o main
```

Or exclusively tell what are the final targets.

Make two objects -----

```
~/MakefileDemo$ make main_1 main
gcc main.c -o main_1
gcc main.c -o main
~/MakefileDemo$ l
main*  main_1*  main.c  Makefile
```

Rules: actions/labels

```
targets: prerequisites  
        command  
        command  
        command
```

Sometimes, a rule does not produce an object.

It only runs some commands.

We call these **targets** 'actions', or 'labels'

```
1 print: main.c  
2      echo 'main.c exists in this directory!'
```

```
~/MakefileDemo$ make print  
echo 'main.c exists in this directory!'  
main.c exists in this directory!
```

Rules: prerequisites

```
1 main: main.c
2         gcc main.c -o main
3 main_1: main.c
4         gcc main.c -o main_1
```

Prerequisites are everything needed for a target.

Also known as “dependencies”.

```
targets: prerequisites
command
command
command
```

Not necessary, when a target doesn't need anything beforehand.

Once a target is being made, 'make' will check all following prerequisites.

If any of them is missing, it will find the **rule** to make that prerequisite.

It traverses through the “dependency tree”, and eventually build the final target(s).

Rules: prerequisites

```
1 main: main.o func.o
2      gcc -o main main.o func.o
3 main.o: main.c
4      gcc -c main.c
5 func.o: func.c
6      gcc -c func.c
```

A prerequisite may also be a target.

```
~/MakefileDemo$ make
gcc -c main.c
gcc -c func.c
gcc -o main main.o func.o
~/MakefileDemo$ l
func.c func.o main* main.c main.o Makefile
```

In this case, the rules for 'main.o' and 'func.o' must exist.

Otherwise, 'make' wouldn't know what to do when it can't find them.

Rules: command

```
targets: prerequisites  
        command  
        command  
        command
```

A set of command tells what to do when the target is required.

They can be shell commands.

Usually, it calls some compilers to make the target.

But it can also look like... also legal

```
1 main: main.c  
2      gcc -o main main.c  
3      echo 'job finished!'  
4      ls -al  
5      mkdir somedir  
6      rm -rf somedir
```


Rules: command

You may notice all executed commands are printed when 'make'.

This is because 'make' will echo anything it does, by default.

To disable this function (silent mode), use '**-s**'

```
~/MakefileDemo$ make -s  
~/MakefileDemo$ l  
func.c  main*  main_1*  main.c  Makefile
```

Rules: "all"

```
1 all: main main_1
2 main: main.c
3      gcc main.c -o main
4 main_1: main.c
5      gcc main.c -o main_1
```

What if we have many targets to make?

One common solution is to define an '**extreme target**', usually named as '**all**'.

Actually, it relies on one basic rule: always try to complete the prerequisites.

'**all**' is the first target, and '**main**' and '**main_1**' are considered as prerequisites.

```
~/MakefileDemo$ make
gcc main.c -o main
gcc main.c -o main_1
```

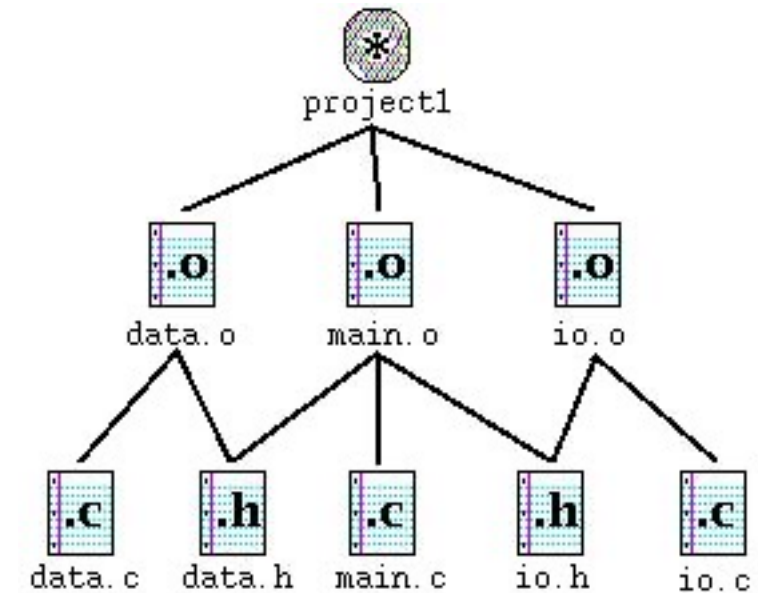
Rules: the common procedure

The mechanism behind 'make'?

1. Figure out the true targets. (final target(s))
2. Based on the prerequisites for 1, build the dependency tree.
 - 1 & 2 is recursive (top to bottom).
3. Build targets based on the commands. (bottom to top)

If 'make' fails to find the rule for some prerequisites, aborts.

So be careful with the script writing, make sure we have enough rules.



Rules: Update

If the final targets are all up to date, they will not be made again.

How do 'make' know they are "up to date"? --- If the prerequisites are all up to date.

Again, a recursive process.

If one of the prerequisites is newer, 'make' will only take care of that.

This avoid useless efforts (only update when needed).

Anyway, '**B**' forces the whole process whether it's up to date or not.

Syntax

Now all the basic rules for 'make' and Makefile is clear.

But there are also plenty of fancy stuff, which would make it easier.

1. Variables
2. Wildcards
3. Implicit Rules
4. Conditional Statements
5. Common Manners

Syntax: Variables

Variables can be used in Makefile script.

Two steps: define and use.

Define a variable directly by **``var=value``**, or **``var:=value``**

Use a defined variable var by **`${var}`**

Makefile will substitute all variables when executing.

That is, **`${var}`** equals to **`value`**

Syntax: Variables

Both variable names and values are string.

No calculation, only substitution.

Can you translate this Makefile script?

```
1 objects = main.o func.o
2 main: ${objects}
3         gcc -o main ${objects}
4 main.o: main.c
5         gcc -c main.c
6 func.o: func.c
7         gcc -c func.c
```


Syntax: Automatic Variables

Automatic variables is a special kind of variables based on each rule.

\$@ = the target

\$^ = all prerequisites

\$< = the leftmost prerequisite

\$? = all prerequisites that are new than the target

Syntax: Automatic Variables

With automatic variables, we can modify the target name without caring commands

```
1 all: main
2 main: main.o func.o
3     gcc -o main main.o func.o
4     echo the target is $@
5     echo prerequisite list: [$^]
6     echo the first prerequisite: $<
```

```
~/MakefileDemo$ make -s
the target is main
prerequisite list: [main.o func.o]
the first prerequisite: main.o
```

Syntax: Wildcards (*)

Wildcard * matches filenames in the file systems, return names separated by space.

Often used to collect a set of files, then assign it to a variable.

Must use in this way: **\$(wildcard ...)**

```
1 source := $(wildcard *.c)
2 print:
3         echo ${source}
```

```
~/MakefileDemo$ make -s
main.c func.c
```

Syntax: Wildcards (%)

Similar rules could be concluded in a “**pattern**”.

Explain by a simple example.

We don't exclusively specify rules for “**main.o**” and “**func.o**” but leave them to match **%.o**

And the ‘%’ of ‘%.c’ and ‘%.o’ refer to the same.

\$^ means “the prerequisites” (**%.c** here)

```
1 all: main
2 main: main.o func.o
3       gcc -o main main.o func.o
4
5 %.o: %.c
6       gcc -c $^
```

Syntax: More about pattern %

Line 5 and Line 6 could be extended as

```
main.o: main.c
        gcc -c main.c

func.o: func.c
        gcc -c func.c

???.o: ????.c
        gcc -c ????.c
```

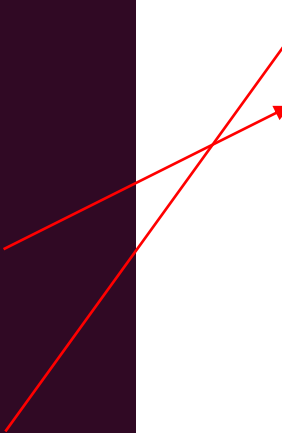
```
1 all: main
2 main: main.o func.o
3         gcc -o main main.o func.o
4
5 %.o: %.c
6         gcc -c $^
```

Syntax: More about pattern %

Only when there is no explicit rule for a target, the pattern rule will be used.

'make' will find the first pattern that matches the target.

```
1 all: main
2 main: main.o func.o
3     gcc -o main main.o func.o
4
5 %.o: %.c
6     gcc -c $^
7     echo $^ is done by pattern
8
9 main.o: main.c
10    gcc -c main.c
11    echo main.c is done explicitly
```



```
~/MakefileDemo$ make -s
main.c is done explicitly
func.c is done by pattern
```

Syntax: 'subst'

`$(subst from,to,text)`

Substitute all 'from' (character) by 'to' in 'text' (string)

```
1 str := abc
2 new := $(subst b,c,$(str))
3
4 print:
5     echo ${new}
```

```
~/MakefileDemo$ make print -s
acc
```


Syntax: "patsubst"

`$(patsubst pattern,replacement,text)`

"pattern" and "replacement" are patterns.

It means to substitute all that match "replacement" by "pattern" in "text".

Syntax: "patsubst"

"patsubst" = pattern substitute

Super important because...

Very useful

```
1 source := $(wildcard *.c)
2 object := $(patsubst %.c,%.o,$(source))
3
4 print:
5     echo $(object)
```

```
~/MakefileDemo$ make print -s
main.o func.o
```

Syntax: Implicit Rules

'make' works more than C/C++ programs, but there are some implicit rules for C/C++.

Not recommended to use, but good to know when reading a Makefile.

Define these implicit variables, added automatically.

The important variables used by implicit rules are:


- `CC` : Program for compiling C programs; default `cc`
- `CXX` : Program for compiling C++ programs; default `g++`
- `CFLAGS` : Extra flags to give to the C compiler
- `CXXFLAGS` : Extra flags to give to the C++ compiler
- `CPPFLAGS` : Extra flags to give to the C preprocessor
- `LDFLAGS` : Extra flags to give to compilers when they are supposed to invoke the linker

Syntax: Implicit Rules

Even the rules for main.o, func.o don't exist the commands for main are not specified, implicit rules provide default making rules for C/C++ programs.

Again, it is better to be explicit.

```
1 CC := gcc
2 CFLAGS := -g
3
4 main: main.o func.o
5
```



```
~/MakefileDemo$ make
gcc -g -c -o main.o main.c
gcc -g -c -o func.o func.c
gcc main.o func.o -o main
```

Syntax: Conditional Statement

Conditional statements are less frequently used in Makefile.

They are mostly used for variables.

Conditional if/else

```
foo = ok

all:
  ifeq ($(foo), ok)
    echo "foo equals ok"
  else
    echo "nope"
  endif
```

Syntax: Conditional Statement

Check if a variable is empty

```
nullstring =  
foo = $(nullstring) # end of line; there is a space here  
  
all:  
ifeq ($(strip $(foo)),)  
    echo "foo is empty after being stripped"  
endif  
ifeq ($(nullstring),)  
    echo "nullstring doesn't even have spaces"  
endif
```

Syntax: Conditional Statement

Check if a variable is defined

`ifdef` does not expand variable references; it just sees if something is defined at all

```
bar =  
foo = $(bar)  
  
all:  
ifdef foo  
    echo "foo is defined"  
endif  
ifdef bar  
    echo "but bar is not"  
endif
```


Syntax: phony targets

A special kind of targets: phony targets

'make' does not treat them as target files, but labels/actions.

Which means, even there is a phony target "clean" AND a file called "clean", 'make' will also execute the commands.

If we do not define "clean" as a phony target, 'make' won't execute it if the file "clean" is up to date!

Syntax: phony targets

The most important and the most commonly used phony target is "**clean**"

Define a phony target by

```
1 all: main
2 main: main.o func.o
3     gcc -o $@ $^
4
5 .PHONY: clean
6 clean:
7     rm main *.o
```

Syntax: Manners

There are some good practice when writing a Makefile.

1. Always use "all" (extreme target)
2. Use automatic variables (\$@, \$^, @<...)
3. Have a phony target "clean"
4. Always explicitly specify the commands, instead of using implicit rules

Practice

Can you design or improve the Makefile for your C/C++ project?

For example, the group project for COMP2432?

Assume the project consists of more than one source file.

M Makefile

```
1 target=PMS
2
3 topdir=./
4 srcdir=$(topdir)SRC/
5 src=$(wildcard *.c)
6 obj=$(patsubst %.c,%.o,$(wildcard $(srcdir)*.c))
7 includedir=$(topdir)INCLUDE/
8 includes=$(wildcard *.h)
9
10 libname=libsched.so
11 libsrcdir=$(topdir)SCHEDULER/
12
13 CFLAG=-g -Wall
14 CFLAG_EXEC=-I$(includedir) -L./ -l$(patsubst lib%.so,%, $(libname)) -Wl,-rpath,./
15
16 all:$(target)
17
18 $(target):$(srcdir) $(obj) $(libname) $(includedir)$(includes)
19     gcc $(obj) -o $@ $(CFLAG) $(CFLAG_EXEC) $(DEB)
20
21 $(libname):$(libsrcdir) $(patsubst %.c,%.o,$(wildcard $(libsrcdir)*.c))
22     gcc -fpic -shared -o $@ $(libsrcdir)*.o $(DEB)
23
24 %.o:%.c
25     gcc $^ -fpic -c -o $@ -I$(includedir) $(DEB)
26
27 .PHONY:clean lib
28 clean:
29     rm -f $(topdir)*.o $(srcdir)*.o $(libsrcdir)*.o *.a *.so *.out $(target)
30 lib:$(obj) $(libname) $(includedir)$(includes)
31     gcc $(obj) -o $@ $(CFLAG) $(CFLAG_EXEC) $(DEB)
32
```

More Resource

There are plenty of resource online.

Dictionary: <https://www.gnu.org/software/make/>

Useful website: <https://makefiletutorial.com/>

Chinese: <https://seisman.github.io/how-to-write-makefile/rules.html>