Make and Makefile

The make utility

- The **`make`** utility is a tool for building software projects
- make uses a descriptor file called a **Makefile** that contains
  - rules to build the *targets*
    - dependency information
    - *commands*
- **Makefile**s are text files and similar in a way to the shell scripts, which are interpreted by the `make` utility.
- Most often, the **Makefile** tells `make` how to compile and link a program

Make Filename

- When you key in make, the make looks for the default filenames in the current directory. For GNU make these are:
  - `GNUMakefile`
  - `makefile`
  - `Makefile`
- If there are more than one of the above in the current directory, the first one according to the above is chosen.
- It is possible to name the makefile anyway you want, then for make to interpret it:

```
make -f <your-filename>
```

Makefile Content

- variables (macros)
- rules (targets) : implicit, explicit
- directives (conditionals)
- # – comments everything till the end of the line
- \ - to separate one command line on two rows

Makefile Content

- A Makefile may have some variables declared for convenience then followed by rules on how to build a given target program.
- Makefile declares variables which are used across all the rules:
  - which compiler options to use,
  - where to look for libraries and include files, etc.
- The rules specify what's needed to build a specific part (target) and how to do it, using shell commands.

Make Variables

- The syntax for declaring and setting a Makefile macro or variable is *varname* = *variable contents*
- To call the variable, use $(*varname*)

```
# Defining the object files
OBJ = main.o example.o

# Linking object files
sample: $(OBJ)
	cc -o sample $(OBJ)
```

Predefined Make Variables

CC	Compiler, defaults to cc.
CFLAGS	Passed to $(CC)
LD	Loader, defaults to ld
LDFLAGS	Passed to $(LD)
$@	Full name of the current target.
$?	Files for current dependency which are out-of-date
$<	The source file of the current (single) dependency

Make Variables

The old way (no variables)

```
my_prog : eval.o main.o
        g++ -o my_prog eval.o main.o
eval.o : eval.c eval.h
        g++ -c -g eval.c
main.o : main.c eval.h
        g++ -c -g main.c
```

A new way (using variables)

```
CC = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
        $(CC) -o my_prog $(OBJS)
eval.o : eval.c $(HDRS)
        $(CC) -c -g eval.c
main.o : main.c $(HDRS)
        $(CC) -c -g main.c
```

Make rules

- rules have the following form:

```
target ... : dependencies ...
<tab>command
<tab>...
<tab>...
```

- A *target* is usually the name of a file that is generated by a program
- A *dependencies* is a file that is used as input to create the target
- A *command* is an action that make carries out

Make Rule

- Makefiles main element is *rule*:

```
target : dependencies
TAB   commands   #shell commands
```

```
my_prog : eval.o main.o
     g++ -o my_prog eval.o main.o

eval.o : eval.c eval.h
     g++ -c eval.c
main.o : main.c eval.h
     g++ -c main.c
```

Make Targets

- Target name can be almost anything:
  - just a name
  - a filename
  - a variable
- There can be several targets on the same line if they depend on the same things.
- A target is followed by
  - a colon “:”
  - and then by a list of *dependencies*, separated by spaced

Make Targets

- The default target `make` is looking for is either `all` or the **first** one in the file.
- Another common target is `clean`
  - Developers supply it to clean up their source tree from temporary files, object modules, etc.
  - Typical invocation is:
    `make clean`

Phony Targets

- Phony targets allow “scripts” to be included in a makefile.
- .PHONY tells Make which targets are not files. This avoids conflict with files of the same name, and improves performance.
- If a phony target is included as a dependency for another target, it will be run every time that other target is required. Phony targets are never up-to-date.

Phony Targets

```
# Naming our phony targets
.PHONY: clean install

# Removing the executable and the object files
clean:
	rm sample main.o example.o
	echo clean: make complete

# Installing the final product
install:
	cp sample /usr/local/.
	echo install: make complete
```

Make Dependencies

- The list of dependencies can be:
  - Filenames
  - Other target names
  - Variables
- Separated by a space.
- May be empty; means “build always”.

Make Dependencies

- Before the target is built:
  - it's checked whether it is up-to-date (in case of files) by comparing time stamp of the target of each dependency; if the target file does not exist, it's automatically considered "old".
  - If there are dependencies that are "newer" than the target, then the target is rebuilt; else untouched.
  - If the dependency is a name of another rule, `make` descends recursively (may be in parallel) to that rule.

Make Actions

- A list of actions represents the needed operations to be carried out to arrive to the rule’s target.
  - May be empty.
- Every action in a rule is usually a typical shell command you would normally type to do the same thing.
- Every command **MUST** be preceded with a **tab**!
  - This is how `make` identifies actions as opposed to variable assignments and targets. Do not indent actions with spaces!

Implicit rules

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules for making an ***.o*** file – from a ***.c*** file, a ***.p*** file, etc. `make` applies the first rule it meets.
- If you have not defined a rule for a given object file, `make` will apply an implicit rule for it.

Example:

Our makefile

```
my_prog : eval.o main.o
   $(CC) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
```

The way `make` understands it

```
my_prog : eval.o main.o
          $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
eval.o : eval.c
          $(C) -c eval.c
main.o : main.c
          $(C) -c main.c
```

Pattern Rules

- A pattern rule is user defined implicit rule
- A pattern rule is a concise way of specifying a rule for many files at once.
- You specify a pattern by using the % wildcard
- The following pattern rule will take any .c file and compile it into a .o file:

```
%.o: %.c
        $(CC) $(CFLAGS) $(INCLUDES) -c <input> -o <output>
```

```
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@
```

Defining Pattern Rules

```
CC = g++
OBJS = eval.o main.o
HDRS = eval.h
%.o : %.c
   $(CC) -c -g $<

my_prog : eval.o main.o
   $(CC) -o my_prog $(OBJS)
$(OBJS) : $(HDRS)
```

<u>Avoiding pattern rules - empty commands</u>

`target: ;`   #Implicit rules will not apply for this target.

Make Directives

Possible conditional directives are:

`if` `ifeq` `ifneq` `ifdef` `ifndef`

All of them should be closed with `endif`.

Complex conditionals may use `elif` and `else`.

Example:

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)          #no tabs at the beginning
else
  libs=$(normal_libs)           #no tabs at the beginning
endif
```

Makefile Example 1

```
CC = gcc
CFLAGS = -g –Wall
OBJFILES= lib.o prog.o
OUTPUT = binary

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o $(OUTPUT)
lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o
prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

.PHONY: clean
clean:
	rm $(OBJFILES) $(OUTPUT)
```

Makefile Example 2

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
        $(CC) $(CFLAGS) $(OBJFILES) -o $(OUTPUT)
%.o: %.c
        # $<: dependency (%.c)
        # $@: target (%.o)
        $(CC) $(CFLAGS) -c $< -o $@
.PHONY: clean
clean:
        rm $OBJFILES) $(OUTPUT)
```

Using makefiles

<u>Running `make`</u>

`>make` – if you want to build the first target of "makefile"

`>make -f` *`filename`* – if the name of your file is not "makefile" or "Makefile"

`>make` *`target_name`* – if you want to make a target that is not the first one

```
> make              #builds first target, i.e., binary
> make binary       #builds specified target, i.e., binary
> make lib.o        #builds specified target, i.e., lib.o
> make prog.o       #builds specified target, i.e., prog.o
> make clean        #builds specified target, i.e., clean
```

Interesting Make Arguments

-d	print debug information
-f <file>	use <file> instead of {mM}akefile
-n	list what would be made; do not execute
-t	‘touch’ files to make them up-to-date; do not execute
-e	env variables override makefile variables

make

https://www.gnu.org/software/make/manual/make.html

- GNU make utility
  - A tool that makes it easy for you to describe how to compile programs to build C/C++ applications
  - Reads a description of a project from a **`Makefile`**
    - You need to describe all the files and their dependencies in the Makefile
- To use the "make" utility, just type "make" at the command prompt
  - By default, make will read and process a **`Makefile`** in your current directory.
  - Otherwise, specify the file name with **`–f`** option
    - **`make –f your_make_file`**

`Makefile`

A **`Makefile`** specifies a set of compilation rules

- in terms of *targets* (such as executables) and their *dependencies* (such as object files and source files)
- The “make” utility goes through the Makefile and follows the chain of dependencies until it reaches the end of the chain and then begins backing out executing the commands found in each target's rule
- **make** looks at the time stamp for each file in the chain and compiles from the point that is required to bring every file in the chain up to date
  - The “**`make`**” utility compiles only those source files that have been changed and the modules that depend upon them

Makefile Format

```
target: dependencies
→a tabcommand
```

- dependencies: names of files depended by the target, i.e:

  ```
  Hello:   Hello.cpp
  ```

- Rule/command: (the how) to construct the target from the dependencies, such as

  ```
  g++ -o Hello Hello.cpp
  ```

- **The first character pre the command line must be a tab**

  The space between the beginning of the line to the command MUST be the tab, not white an empty spaces

A Simple Makefile

- Note: the line index is not part of the Makefile

```
1 #Makefile
2 GCC = gcc
3 OBJS  = foo.o bar.o baz.o
4 CLFAGS = -Wall -O2
5 LDLIBS = -L./ -lbar

6 prog: $(OBJS)
       $(GCC) -o prog $(OBJS) $(LDLIBS)
7 foo.o: foo.c
       $(GCC) $(CFLAG) -c foo.c
8 bar.o: bar.c
       $(GCC) $(CFLAG) -c bar.c
9 baz.o: baz.c
       $(GCC) $(CFLAG) -c baz.c

10.PHONY: install clean
11 install:
       install -m 755 foo $HOME/local/bin
12 clean:
       rm *.o; rm foo
```

Makefile Explained

- L1: comment
- L2-L5: define variables (macro) OBJS, LDLIBS, GCC
- L6-L9: definition of **compilation rules**
  - It states that target `prog` depends on (or is built from) the object files whose names are contained in variable **`OBJS`** (called **dependency** list)
  - command line after that tells how to build the target from the dependency list, **the first character in the command line must be a tab**
- L10, .PHONY:
  - tells make that install and clean are not target files to avoid a conflict with a file of the same name
  - And there is no dependents for the phony target, so it will always be executed when the target is requested
- L11-12: install and clean are phony targets

Phony Targets

- A phony target is one that is not really the name of a file
- It is just a name for a recipe to be executed when you make an explicit request.
- There are two reasons to use a phony target
  - to avoid a conflict with a file of the same name,
  - to improve performance.
- Command: `make clean`
  - Will execute the recipe: `rm *.o; rm fo`
- Command: `make install`
  - Will execute the recipe: `install –m 755 foo $HOME/local/bin`
    - `install:` `copy files and set attributes`

For more detail:

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

“Implicit Rules”

- Built-In Rules
  - Compiling C programs file.o

    `$(CC) $(CFLAGS) -c`

  - Compiling C++ programs
    - `$(CXX) $(CXXFLAGS) -c`
- Pattern Rules (prefix %)
  - `%.o:%.c`   How to make a .o file from a .c file

https://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html#Implicit-Rules

Automatic variables--special symbols

```
.c.o:
     $(GCC) $(CFLAG) -c $<
```

⇕

```
%.o: %.c
     $(GCC) $(CFLAG) -c $<
```

```
foo: $(OBJS)
     $(GCC) -o $@ $(OBJS) $(LDLIBS)
```

- **The inference rule**
  - `.s1.s2:` describes how to build a target that is appended with `.s2` with a prerequisite that is appended with `.s1`.
- `.SUFFIXES: .o:.c`
- `$*` ➔ current target without the extension
- `$<` ➔ is a dependent file (full name of the prerequisite file)
- `$@` ➔ represents the full target name of the current target

“make” Inference Rules

```
6 foo.o: foo.c
        $(GCC) $(CFLAG) -c foo.c
7 bar.o: bar.c
        $(GCC) $(CFLAG) -c bar.c
8 baz.o: baz.c
        $(GCC) $(CFLAG) -c baz.c
```

Look lines 7-9, the rules to make OBJECT file, they are very similar

```
<filename>.o : <filename>.c
$(GCC) $(CFLAG) -c <filename>.c
```

```
.c.o:
     $(GCC) $(CFLAG) -c $<
```

```
6 foo: $(OBJS)
        $(GCC) -o foo $(OBJS) $(LDLIBS)
```

```
5 foo: $(OBJS)
        $(GCC) -o $@ $(OBJS) $(LDLIBS)
```

File: Makefile with variables (macros)

```
#Makefile, the indent in the rules are always a TAB
PROGRAM        := qemployee
CXX            := c++
SRC            := employee.cpp main.cpp
OBJS           := employee.o main.o
LIBDIRS        := ../lib
INCLUDEDIRS    := ../include
LBFLAGS        := -L$(LIBDIRS)
CXXFLAGS       := -Wall -O2 -I$(INCLUDEDIRS)
$(PROGRAM):$(OBJS)

        $(CXX)  $(OBJS) -o $(PROGRAM) $(LBFLAGS)
employee.o: employee.cpp
        $(CXX) $(CXXFLAGS) -c employee.cpp -o employee.o
main.o: main.cpp
        $(CXX) $(CXXFLAGS) -c main.cpp  -o main.o
clean:
        rm *.o; rm employee
install: empolyee
       install -m 755 employee $HOME/local/bin
.PHONY: install clean
```

References

- An Introduction to GCC

  http://www.network-theory.co.uk/docs/gccintro/index.html

- GNU Debugger

  http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

- GNU “make”

  http://www.gnu.org/software/make/manual/make.html