Make and Makefile

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

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	Files for current dependency which are out-of-date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)
```

Avoiding pattern rules - empty commands

`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