GNU Make

A simple and hands-on introduction through examples

Operating Systems / MIEIC

U.PORTO

Pedro Silva

March 19, 2019

What is GNU Make?

- A build automation tool that
  - Eases the building process of executables and libraries from source code
  - Reads the set of building rules from a file (known as **makefile**)
  - Supports incremental builds (only builds targets whose sources have changed)
  - Enables to define which targets are to be built
  - Supports variable definition through command line arguments

```
# reads the set of rules from Makefile [or makefile] file (default)
# the default target is the first one whose name does not start with '.'
$ make

# reads the set of rules from MyMakefile file
$ make -f MyMakefile

# builds targets mytarget1 and mytarget2 [supports multiple targets]
$ make mytarget1 mytarget2

# defines variable X as 1 (target and makefile are the default ones)
$ make X=1
```

- More details in GNU make's manual (how make processes a makefile)

What is a GNU Makefile?

- A file consisting of a set of rules
  - The **target** (left side) is **what** is being built
  - The **prerequisites** (right side) are the list of dependencies to build it **from**
  - The **recipe** is the list of commands defining **how** to build it

```
# => each command starts with a tab character ('\t', not spaces) <=

# build executable mytarget from mysource.c
# [any changes on myheader.h file will also trigger a rebuild]
mytarget: mysource.c myheader.h
  gcc -Wall -Werror -pedantic mysource.c -o mytarget
```

- Interpreted using GNU make (there are other variants such as BSD)

```
# makefile is file Makefile; mytarget is the first (and only) target
$ make -f Makefile mytarget
```

- More details in GNU make's manual (what a rule looks like)

Recipe Echoing

- By default, each command within the **recipe** is echoed

```
mytarget: mysource.c myheader.h
  gcc -Wall -Werror -pedantic mysource.c -o mytarget
```

```
$ make
gcc -Wall -Werror -pedantic mysource.c -o mytarget
```

- Recipe echoing can be suppressed by prefixing the command with '@'

```
mytarget: mysource.c myheader.h
  @gcc -Wall -Werror -pedantic mysource.c -o mytarget
```

```
$ make
# no recipe echoing
```

- More details in GNU make's manual (recipe echoing)

Phony Target

- The problem (**make** expects a regular target to be a file)

```
clean:
  # -f option suppresses the error message if the file does not exist
  rm -f mytarget
```

```
make clean && touch clean && make clean
rm -f mytarget
make: 'clean' is up to date.
# clean's recipe is not executed twice because touch creates 'clean' file
```

- A phony target is a target that is not really the name of a file
  - Avoids conflicts with a file with the same name
  - The rule will not create a target, therefore it will always be executed

```
.PHONY: clean
clean:
  rm -f mytarget
```

```
make clean && touch clean && make clean
rm -f mytarget
rm -f mytarget
```

- More details in GNU make's manual (phony targets)

Very Simple Makefile Example

- The Makefile

```
.PHONY: all clean
# as common practice, the first target is 'all' [builds all intended targets]
all: mytarget

mytarget: mysource.c myheader.h
  gcc -Wall -Werror -pedantic mysource.c -o mytarget

clean:
  rm -f mytarget
```

- Examples of usage

```
make clean && make && make
rm -f mytarget
gcc -Wall -Werror -pedantic mysource.c -o mytarget
make: 'mytarget' is up to date.
# there is no need to rebuild mytarget a second time (no changes)
```

```
touch mysource.c && make
gcc -Wall -Werror -pedantic mysource.c -o mytarget
# mytarget is rebuilt when mysource.c was "modified" by touch command
# [make relies on file modification time to flag which sources have changed]
```

Enabling Incremental Builds (1)

- Defining intermediate targets enables to avoid redoing unnecessary steps
  - Aimed at multiple source files and external dependencies (e.g. libraries)
- Understanding Compilation Stages
  - Preprocessor Stage

    The C/C++ preprocessor expands macros and processes a set of directives such as #include (header files) and #ifdef (conditional compilation).
  - Compiler Stage

    The compiler takes the preprocessor's output and translates it into assembly source code.
  - Assembler Stage

    The assembler takes the assembly source code, either generated by the compiler or written by a developer, and produces machine code. The output is stored in an object file.
  - Linker Stage

    The linker takes object files and libraries as input, performs several operations such as resolving references to external symbols, and produces a binary file (usually an executable).
  - The default file extension list is available in GCC's manual (overall options)

Enabling Incremental Builds (2)

- Each source code file has a corresponding object file
  - Any changes on a source file only requires rebuilding that object file
- Object files (and any libraries) are the only required intermediate targets
  - There is no need to save C/C++ preprocessor and compiler stages' output
- Makefile

```
.PHONY: all clean

all: mytarget

# binary is generated from object files to avoid compiling all source files
# [relevant when there are multiple object files or external dependencies]
mytarget: mysource.o
  gcc -Wall -Werror -pedantic mysource.o -o mytarget

# dependency rule for building mysource.o object file
mysource.o: mysource.c myheader.h
  gcc -Wall -Werror -pedantic -c mysource.c -o mytarget.o

# removes target and any object files
clean:
  rm -f mytarget *.o
```

Using Automatic Variables and Pattern Rules

```
.PHONY: all clean

all: mytarget

# $@: The file name of the target of the rule.
# $<: The name of the first prerequisite.
# $^: The names of all the prerequisites (without duplicates).

mytarget: mysource.o
  gcc -Wall -Werror -pedantic $^ -o $@

# the pattern rule defined below will be used as the recipe
mysource.o: mysource.c myheader.h

# pattern rule for compiling C source code files (% is a wildcard)
%.o: %.c
  gcc -Wall -Werror -pedantic -c $< -o $@

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

- More details in GNU make's manual (automatic variables)
- More details in GNU make's manual (pattern rules)

Using Variables (1)

- GNU make provides the following set of assignment operators
  - VAR = *expression*

    Recursively expanded variable. The expression is only evaluated when it is used (deferred).
  - VAR := *expression* or VAR ::= *expression*

    Simply expanded variable. The expression is evaluated when it is defined (immediate).
  - VAR ?= *expression*

    Conditional variable assignment. The variable is only set if it is not yet defined.
  - VAR += *expression*

    Appending assignment. The expression evaluation result is appended to the variable.
  - VAR != *shell command*

    Shell assignment. Sets the variable to be the output of the shell command.

  - More details in GNU make's manual (how make reads a makefile)
  - More details in GNU make's manual (the two flavors of variables)
  - More details in GNU make's manual (appending more text to variables)
- Variables can be used through $(VAR) notation

Using Variables (2)

- Makefile

```
CC=gcc
CFLAGS=-Wall -Werror -pedantic
TARGET=mytarget
SRCS=mysource.c
# replaces sources' extension .c by .o (object file)
OBJS=$(SRCS:.c=.o)

# other variables for libraries and include paths could also be defined

.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) $^ -o $@

mysource.o: mysource.c myheader.h

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

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

Automatic Dependency Lists

- Manually defining the list of dependencies is, above all, error-prone
- GCC and clang provide automatic dependency file generation
  - More details in GCC's manual (preprocessor options)
- Makefile (variable declaration is omitted)

```
.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) $^ -o $@

# -MMD generates a file (*.d) for the list of dependencies [user headers only]
%.o: %.c
  $(CC) $(CFLAGS) -MMD -c $< -o $@

# remove also dependency list files (*.d)
clean:
  rm -f $(TARGET) *.o *.d

# include the list of dependencies generated for each object file
-include $(SRCS:.c=.d)
```

Collecting the Output of Each Compilation Stage

- GCC/clang provides *-save-temps* option to collect each stage's output
  - Alternatively, a pattern rule could be defined for each stage
  - More details in GCC's manual (developer options)

- Makefile (variable declaration is omitted)

```
.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) $^ -o $@

# -save-temps generates preprocessor (*.i) and compiler (*.s) output files
%.o: %.c
  $(CC) $(CFLAGS) -MMD -save-temps -c $< -o $@

# remove also preprocessor (*.i) and compiler output (*.s) files
clean:
  rm -f $(TARGET) *.o *.d *.i *.s

-include $(SRCS:.c=.d)
```

Conditional Statements (1)

- GNU make supports conditional statements
  - More details in GNU make's manual (conditional parts of makefiles)
- Makefile (variable declaration is omitted)

```
.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) $^ -o $@

# flag/option -save-temps is only added if DEV_INFO is defined
ifdef DEV_INFO
DEV_FLAGS =-save-temps
endif

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

clean:
  rm -f $(TARGET) *.o *.d *.i *.s

-include $(SRCS:.c=.d)
```

Conditional Statements (2)

- Compilation **without** each stage's output

```
make clean && make && ls
rm -f mytarget *.o *.d *.i *.s
gcc -Wall -Werror -pedantic  -MMD -c mysource.c -o mysource.o
gcc -Wall -Werror -pedantic mysource.o -o mytarget
Makefile  myheader.h  mysource.c  mysource.d  mysource.o  mytarget
# flag/option -save-temps is not present (no *.i and *.s files)
```

- Compilation **with** each stage's output
  - DEV_INFO variable could be set with any other value
    (The makefile only checks if the variable is defined)

```
make clean && make DEV_INFO=1 && ls
rm -f mytarget *.o *.d *.i *.s
gcc -Wall -Werror -pedantic -save-temps -MMD -c mysource.c -o mysource.o
gcc -Wall -Werror -pedantic mysource.o -o mytarget
Makefile  myheader.h  mysource.c  mysource.d  mysource.i  mysource.o  mysource.s  m
# flag/option -save-temps is present (generates *.i and *.s files)
```

Using Built-in Functions

- A GNU make function is invoked as $(*function param,param,...*)
- GNU make provides several built-in functions
  - $(wildcard *pattern ...*)

    For each file name pattern, provides a space-separated list with the names of existing files that match it (may have duplicates); the output of non-matching patterns is omitted.

  - $(patsubst *pattern,replacement,text*)

    Replaces (**replacement**) whitespace-separated words in **text** that match the **pattern**.

  - $(foreach *var,list,text*)

    Repeatedly performs a different substitution of variable **var**, using each single value within the **list**, when evaluating a piece of **text**.

  - $(call *function,param,param,...*)

    Enables to create new parametrized functions: a complex expression written as the value of a variable that can be expanded with different values (arguments) using **call** function.

  - $(eval *expression*)

    Allows to define non-constant makefile constructs that result from the evaluation of other variables and functions. The expanded results can define new variables, targets, etc.

  - More information in GNU make's manual (functions for transforming text)

User Defined Functions

- A function is a complex expression written as the value of a variable
  - Variables $(0) [function name], $(1), ... are function parameter placeholders

```
# function to dynamically generate targets
generate_target = $(eval $(call make_target,$(1)))
```

  - The expression can be also a multi-line variable (*define* ... *endef* syntax)

```
# $$ will get evaluated into $ to keep variables $^ and $@ in the recipe
define make_target
$(1): $(1).o
  $(CC) $(CFLAGS) $$^ -o $$@
endef
```

- Invoked using function **call**
  - Function parameter placeholders are expanded using the arguments given

```
# create a rule for each required target
# (doing so, autocomplete feature of make targets works as expected)
$(foreach TARGET,$(TARGETS),$(call generate_target,$(TARGET)))
```

```
# create a target for each source file (filename without .c extension)
TARGETS = $(patsubst %.c,%,$(wildcard *.c))
```

An Example Makefile for Practical Classes (1)

```
CC=gcc
CFLAGS=-Wall -Wextra -Werror -pedantic
TARGETS = $(patsubst %.c,%,$(wildcard *.c))

define make_target
$(1): $(1).o
  $(CC) $(CFLAGS) $$^ -o $$@
endef

.PHONY: all clean
all: $(TARGETS)

ifdef DEV_INFO
DEV_FLAGS = -save-temps
endif

$(foreach TARGET,$(TARGETS),$(eval $(call make_target,$(TARGET))))

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

clean:
  rm -f $(TARGETS) *.o *.d *.i *.s

-include $(TARGETS:=.d)
```

An Example Makefile for Practical Classes (2)

- This makefile enables to
  - Save time when compiling without disabling warnings (treated as errors)
  - Dynamically create a target for each C source file (no changes needed)
  - Build all targets at once (meaningful executable file names; no *a.out* files)
  - Collect the C/C++ preprocessor (*.i) and compiler (*.s) outputs
- Examples of usage

```
make clean && ls && make DEV_INFO=1 p1 && ls
```

```
rm -f  p2a  p1 *.o *.d *.i *.s
Makefile  p1.c  p2a.c
gcc -Wall -Wextra -Werror -pedantic -save-temps -MMD -c p1.c -o p1.o
gcc -Wall -Wextra -Werror -pedantic p1.o -o p1
Makefile  p1  p1.c  p1.d  p1.i  p1.o  p1.s  p2a.c
```

```
make clean && ls && make
```

```
rm -f  p2a  p1 *.o *.d *.i *.s
Makefile  p1.c  p2a.c
gcc -Wall -Wextra -Werror -pedantic  -MMD -c p2a.c -o p2a.o
gcc -Wall -Wextra -Werror -pedantic p2a.o -o p2a
gcc -Wall -Wextra -Werror -pedantic  -MMD -c p1.c -o p1.o
gcc -Wall -Wextra -Werror -pedantic p1.o -o p1
```