# [Software Development]

# Makefiles

Davide Balzarotti

Eurecom – Sophia Antipolis, France

# **Software Development Tools**

- 1. Configuring and Building the program
  - GCC
  - Makefiles
  - Autotools
- 2. Writing and managing code
- 3. Packaging and Distributing the application
- 4. Debugging and Profiling

# Why Makefiles

- Programs usually contains multiple source files
  - Written by different people
  - Often organized in subdirectories
  - With complicated dependency
     (to compile X you first have to compile Y, that requires to compile the library Z..)

#### Problems:

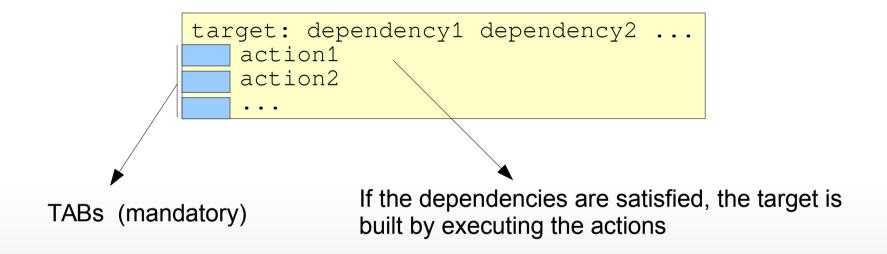
- Recompiling the entire project for each small change it is very inefficient and may requires long compilation time
- Maintaining complex projects quickly becomes a nightmare

### **Make and Makefiles**

- Makefiles date back to 1977 but are still the most widely used tool to manage the build process
  - Different versions exist: e.g., GNU make, bsd make, Microsoft nmake
- A makefile is a script that describes:
  - The project structure: the files and the dependency between them
  - How to derive the target program from each of its dependencies
  - Instructions to compile each file
- The instructions inside a makefile are interpreted and executed by the make program
  - By checking the dependencies, make only recompiles what needs to be recompiled

### Structure of a Make File

A makefile is a text files containing one or more rules



- Makefiles can also contains variable (macro) assignments:
  - Definition: CC = gcc
  - Use: echo \${CC} or echo \$(CC)
  - By convention, all variables in a makefile are uppercase

# **A Simple Makefile**

```
all: func.o main.o
                                            The target "all" depends on the targets
    qcc func.o main.o -o main
                                            func o and main o
func.o: func.c func.h
    qcc -q -c func.c
                                         The target "func.o" depends on func.c
main.o: main.c func.h
    qcc -q -c main.c
                                           Actually, any .o depends by default on
clean:
                                            the corresponding .c file
    rm - f *.o
                                             (so func.c could be omissed)
    rm main
                                           The target "clean" does not have any
                                            dependency
```

### **How It Works**

- The make command first look for a file named makefile or Makefile in the current directory
- Then, it parses the file content and construct the dependency tree
- Then it checks if any of the target prerequisites does not exist or it is newer than the target itself (by checking file timestamps)
  - If so, it first executes the prerequisite targets (which, of course, can have other dependencies and so on..)
  - If the target is not a filename, make cannot determine its timestamp and therefore it always executes it
- For each action in a target, make prints the action and then execute it (in a <u>separate subshell</u> for each action command)

# **A Simple Makefile**

```
# This is a comment
all: func.o main.o
   gcc func.o main.o -o main

func.o: func.c func.h
   gcc -g -I. -c func.c

main.o: main.c func.h
   gcc -g -I. -c main.c

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

- make target: executes the action associated with target (and the required dependences)
- make : executes the first target specified in the makefile

```
> make
gcc -g -I. -c func.c
gcc -g -I. -c main.c
gcc func.o main.o -o main
> touch func.c
> make
gcc -g -I. -c func.c
gcc func.o main.o -o main
> make clean
rm -f *.o
rm main
```

# More on Variable Assignment

- There are four way to assign a variable:
  - := (simple assignment)
    - = (recursive assignment)
      - the right side of the assignment expression is not evaluated immediately but when the variable is used
    - Example:

```
COMPILE = ${CC} ${CFLAGS}
```

- ?= (conditional assignment)
  - The assignment is performed only if the variable did not have a value yet
- += (append)
  - Append text to an existing variable

### **Patterns**

#### Defining patterns

- The percent sign, %, can be used to perform wildcard matching to write more general targets
- When a corresponding % appears in the dependencies list, it is replaced by the same string of text that assumed in the target
- Example:

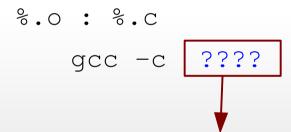
```
%.o: %.c

gcc -c ????
```

### **Patterns**

#### Defining patterns

- The percent sign, %, can be used to perform wildcard matching to write more general targets
- When a corresponding % appears in the dependencies list, it is replaced by the same string of text that assumed in the target
- Example:



PROBLEM: How can I reference the filename in the action?

# **Special Variables**

#### Special variables

- \$@ full target name of the current target
- \$? evaluates to the list of prerequisites that are newer than the current target
- \$\* evaluates to the current target name with its suffix deleted (in practice, the value of the % wildcard in the target)
- \$< The name of the first dependency</li>
- \$^ The names of all the dependencies, with spaces between them

```
main.o: main.c operation.h gcc -c $< -o $@
```

# **Using Shell Wildcards in Makefiles**

 Suppose you want to express that the foo target is made (i.e., depends) of all the object files in the current directory

```
objects = *.o
foo : $(objects)
cc -o foo $(CFLAGS) $(objects)
```

 Each existing '.o' file becomes a dependency of foo and will be recompiled if necessary

# Using Shell Wildcards in Makefiles

Suppose you want to express that the foo target is made (i.e., depends) of all the object files in the current directory

```
objects = *.0
foo: $(objects)
      cc -o foo $(CFLAGS) $(objects)
```

- Each existing '.o' file becomes a dependency of foo and will be recompiled if necessary
- But what happen if there are no .o files?
  - Since the wildcard does not match any file, it is left as it is
  - Therefore foo will depend on a file named '\*.o'
  - Since '\*.o' does not exists, make will raise an error because it does not know how to build it

# **Functions**

- Functions allow text processing inside a makefile
- Usually used to
  - compute a file list to operate on
  - compute the commands to use
- Functions are invoked with the following syntax

```
$(function_name arg1, arg2, ... argN)
```

- function\_name must be one of the pre-defined functions
  - Check Makefile documentation for the list of functions
  - It is not possible to define new functions

### **Useful Functions**

- \$ (subst from, to, text)
  - Substitute <from> with <to> in <text>
- \$ (patsubst pattern, replacement, text)
  - Finds whitespace-separated words in <text> that match <pattern> and replaces them with <replacement>

```
$(patsubst %.c,%.o,foo.c bar.c) → foo.o bar.o
```

- \$ (filter pattern...,text)
  - Removes all whitespace-separated words in text that do not match any
    of the pattern words, returning only matching words

### **Useful Functions**

- \$ (shell command)
  - Takes as argument a shell command and returns the output of the command

```
contents := $(shell cat foo)
```

- \$ (wildcard pattern)
  - Return a space-separated list of names of existing files that match the pattern
- \$ (dir names...)
  \$ (notdir names...)
  - Extract and return the directories (or the filenames) from a list of file names

```
$(dir dir1/foo.c dir2/bar.h) → dir1 dir2
$(notdir dir1/foo.c dir2/bar.h) → foo.c bar.h
```

### **Useful Functions**

- \$ (addprefix prefix, names...)
  - Prepend prefix> at the beginning of each file name

```
$(addprefix src/, foo bar) → src/foo src/bar
```

- \$ (basename names...)
  - Extracts all but the extension of each file name

```
$ (basename src/foo.c bar.x new) → src/foo bar new
```

- \$ (addsuffix suffix, names...)
  - Append a suffix at the end of each file name

```
$(addsuffix .c, foo bar) → foo.c bar.c
```

### **Action Modifiers**

- Each action can be modified by pre-pending one of the following prefixes
  - (dash) any error found while executing the command is ignored
    - By default, make stops if one of the action returns an exit status different than zero
    - With make prints out a message with the status code the command exited with, and says that the error has been ignored
    - Example:
      - -rm file # delete the file but does not stop if it does not exist
  - (at) the command is not written to standard output before it is executed
    - Example: @echo "hello" # print hello, without printing the echo command

```
CC = gcc
                          # compilator to use
CC_OPT = -02 - g - Wall # compilation parameters
INCLUDES = -I.
                          # path to the .h files
CFLAGS = \$\{CC\_OPT\}
OBJS = func.o main.o # list of object file to build
all: ${OBJS}
   ${CC} ${CFLAGS} ${OBJS} -o main ${INCLUDES}
8.0: 8.C
   ${CC} -c ${CFLAGS} $*.c ${INCLUDES}
clean:
   @echo "Cleaning..."
   rm -f *.o
   rm main
```

# **Phony Targets**

- A target does not necessarily need to be a filename
  - But if there is a file in the current directory with the same name of the target, make gets confused:

```
> make clean
rm -f *.o
rm main
> touch clean
> make clean
make: 'clean' is up to date.
```

 A way to explicitly specify that a target is NOT a filename is to declare it as a phony target

```
.PHONY:clean

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

### **Nested Makefiles**

```
subsystem:
cd subdir && $(MAKE)
```

- Useful for large systems containing many directory, each with its own makefile and you would like the containing directory's makefile to run make on the subdirectory
- Note the use of the \$(MAKE) variable instead of calling the make program
  - It ensures that make is always executed (also if the user is just "simulating" the makefile with -t or -n options)

# **Useful stuff**

- If you want to know the commands that will be executed to generate a file, run make with -n
- If you want to change Makefile variables from the command line, specify them after the target, i.e.

```
make test.o CC=gcc CFLAGS="-g -00"
```