

MAKE UTILITY

## What is Make?

- Managing Projects with make – Oram & Talbott – O'Reilly
- **make** is a command generator
- Using a description file and some general templates, it creates a sequence of commands for execution by the UNIX shell
- These commands commonly relate to the maintenance of the files comprising a software development project
- **make** is used to sort out dependencies among files – modification of one or more files may necessitate recompiling/relinking – this process is to be repeated many times over the life cycle of a project – **make** simplifies the process

## Simple Makefile

- `% make program` – indicates that the latest version of `program` is to be made – if `program` is an executable file → perform all the necessary compilation and linking required to create this file
- Instead of typing all the commands (repeatedly) by hand, automate them by `make`
- `program` is called the *target* – if it is to be built from one or many files, these files are called as *prerequisites/dependents*
- The definition of target and dependents is recursive

## Simple Makefile

```
program: main.o class.o
    g++ -o program main.o class.o

main.o: main.cc
    g++ -c main.cc

class.o: class.cc class.h
    g++ -c class.o class.cc
```

## Explanation

- 3 entries in the simple makefile
- Each entry consists of a line containing a colon (the *dependency* or *rule line*, and one or more *command lines* beginning with a *tab*.
- The left side of a dependency line (before the colon) is a target; and the right side contains that target's prerequisites
- The tab-indented command lines show how to build the target from the prerequisites
- `% make program` – will create the target `program` – with minimum recompilations
- Creating a target consists of a chain of commands that must be issued in the correct order – generally, `make` is told to build the last file in that chain – *make* will trace back through all the dependencies and execute the necessary commands, until the target is built – backward-chaining

## Invoking Make

- Assumptions – all the necessary files (.h, .cc) and the description files are in the same directory – The description files are named either **makefile** or **Makefile**
- % **make target** – each command line required to build the target is echoed on the terminal and executed – if some files are up to date, their commands are ignored
- If no prerequisite files were modified/removed since the last invocation of make (for the same target), make issues '*target*' is up to date message
- % **make** – the first target in the description file is made (along with all the necessary pre-requisites)
- % **make -f file\_name** – if the description file is called **file\_name** (instead of **makefile** or **Makefile**)

## Basic Syntax Rules

- Begin every command by the initial tab – common error!
- Do NOT start any other line (e.g., dependency line or comment) with a tab
- Tabs can be used freely anywhere except the first characters
- To check tabs in a makefile – `% cat -v -t -e makefile - -v` and `-t` options will cause all the tabs to appear as `^I` and `-e` option places a dollar sign at the end of each line
- A line can be continued by placing a backslash `\` at the end – *place the backslash right before the newline, without any whitespace in between*
- `#` indicates comments – lines starting with `#` are ignored by make
- `clean: /bin/rm -f core *.o` – allows the programmer to delete the temporary files (such as `*.o` and `core`)  
`% make clean` will perform the necessary task

## Macros

- Description files generally contain repetitive actions – in a large project that would lead to many repeated text lines → use macros
- `name = text string` – defines a macro – `$(name)` or `${name}` uses the macro
- `OBJS = circle.o square.o shape_main.o`
- `shape_exec: $(OBJS)`  
`g++ -o shape_exec $(OBJS)`
- Invocation of `% make shape_exec` will result in the expansion of the macro – `OBJ`  
`% g++ -o shape_exec circle.o square.o shape_main.o`

## Syntax Rules

- A macro definition line contains an equal sign
- # starts a comment
- Use backslash for the continuation of the macro definition on the next line
- NO TABS are permitted before a macro name – to differentiate between a macro definition and a command/dependency line
- Macro names, by convention, are in UPPERCASE – Avoid shell metacharacters like \ and >
- The order of macro definitions is immaterial – flexibility in writing macros
- A macro name cannot be redefined

## Internally Defined Macros

- `$(CC)` – recognized by **make** as the C/C++ compiler
- `$(LD)` – recognized by **make** as the linker

- `CC = g++`

```
CFLAGS = -g -c
```

```
circle.o: circle.cc shape.h circle.h
```

```
    $(CC) $(CFLAGS) circle.cc
```

```
=> circle.o : circle.cc shape.h circle.h
```

```
    g++ -g -c circle.cc
```