UNIX Window Systems and Development: g++ and make

CS 2204
Class meeting 7
Created by Doug Bowman, 2001
Modified by Mir Farooq Ali, 2002
Why window systems?

- Increased usability due to
  - Visibility
    - Graphical representation of programs
    - See multiple environments at once
  - Direct manipulation
- Enables powerful graphics programs
Window systems and UNIX

- Most UNIX users can be considered experts, and are fiercely protective of the command line.
- However, all current UNIX systems have a built-in window system - the advantages are inescapable.
X Windows

- Practically all UNIX window systems are based on X Windows
- The standard version is called X11
- Very complex system with many parts
- Basically, X11:
  - Manages the screen space
  - Draws simple graphics
  - Assigns rectangular regions to various programs
X’s client-server architecture

- X is actually meant to work over the network
- X server: software that runs on the machine where the program’s output will be displayed
- X client: program running on the same or another machine
- Client sends drawing and other X commands to the server, which displays the results
Historical use of X

- Users sat at “X terminals” - graphical terminals that only knew how to run an X server
- They logged in to other UNIX machines remotely and ran X clients there
- This gave users the benefits of a window system without the need for a full-featured computer on every desk
Features of X

- Transparent remote execution
- Gives programs their own virtual screen
- Includes important windowing concepts:
  - Window damage
  - Window reveal events
  - Backing store
- X11 programs are highly portable
Window managers

- Not part of X11 itself
- Run on top of X11
- Place borders, decorations on windows
- Handle input from users
- There are many, many choices with different “look & feel”
Desktop environments

- Yet another layer, running on top of window managers
- Complete the desktop metaphor with:
  - Iconic access to files and directories
  - Overall system menus / toolbars
  - Trash can
  - etc.
- GNOME, KDE are different examples
Overview of development process

- Creation of source files (.c, .h, .cpp)
- Compilation (e.g. *.c → *.o) and linking
- Running and testing programs
Development tools in UNIX

- Creation of source files (.c, .h, .cpp)
  - Text editors (e.g. vi)
  - Revision control systems (e.g. rcs)
- Compilation (e.g. *.c → *.o) and linking
  - Compilers (e.g. gcc)
  - Automatic building tools (e.g. make)
- Running and testing programs
  - Debuggers (e.g. gdb)
- Integrated development environments (IDEs)
Compiling with `g++`

- GNU C++ compiler
- Performs one or more of the following:
  - C++ pre-processing
  - Compilation
  - Linking
Basic \texttt{g++} examples

- \texttt{g++ hello.cc} (compile \texttt{hello.cc}, produce executable \texttt{a.out})
- \texttt{g++ -o hello hello.cc} (compile \texttt{hello.cc}, produce executable \texttt{hello})
- \texttt{g++ -o hello hello.cc other_functions.cc} (compile \texttt{hello.cc} and \texttt{other_functions.cc}, produce executable \texttt{hello})
Using intermediate files

- From any source file, you can produce an object file to be linked in later to an executable

```
g++ -c hello.cc
```
```
g++ -c other_functions.cc
```
```
g++ -o hello hello.o other_functions.o
```
Other important *g++* options

- `-g`: include debugging symbols in the output
- `-l<name>`: include a library called `libname.a`
Include and library paths

- There are default directories in which g++ looks for include files and libraries
- \(-I<path>:\) also look for include files in this directory
- \(-L<path>:\) also look for library files in this directory
Defines in g++

- Often programs contain conditional parts based on defines:
  
  ```c
  #ifdef DEBUG
  printf("value of var is %d", var);
  #endif
  
  - You can set preprocessor defines on the command line
  ```
  
  ```c
  g++ -DDEBUG -o prog prog.c
  ```
Using *make* in compilation

- With medium to large software projects containing many files, it’s difficult to:
  - Type commands to compile all the files correctly each time
  - Keep track of which files have been changed
  - Keep track of files’ dependencies on other files
- The *make* utility automates this process
Basic operation of `make`

- Reads a file called `Makefile`, which contains rules for building a “target”
- If the target depends on a file, then that file is built
- If that file depends on a third file, then the third file is built, and so on…
- Works backward through the chain of dependencies
- Targets only built if they are older than the files they depend on
Basic Makefile example

```
program : main.o iodat.o dorun.o
    g++ -o program main.o iodat.o dorun.o
main.o : main.cc
    g++ -c main.cc
iodat.o : iodat.cc
    g++ -c iodat.cc
dorun.o : dorun.cc
    g++ -c dorun.cc
```
Types of lines in Makefiles

- Dependency or rules lines
- Commands
- Macro assignments
- Comments
Dependency/rules lines

- Specify a target and a list of prerequisites (optional) for that target

  target : prereq1 prereq2 prereq3 ...
Command lines

- Follow dependency lines
- MUST start with a TAB!
- Any command that can be run in the shell can be placed here

```
target : prereq1 prereq2
  command1
  command2
```

- Special variables in commands:
  - `@` represents the target
  - `@` represents prereqs that are newer than target
Macro (variable) assignments

- You can use macros to represent other text in a Makefile
  - Saves typing
  - Allows you to easily change the action of the Makefile

- Assignment:
  \[ \text{MACRONAME} = \text{macro value} \]

- Usage: \$ \{MACRONAME\}
Comments and other Makefile notes

- Comments begin with a ‘#’
- Can be placed at the beginning of a line or after a non-comment line

- Lines that are too long can be continued on the next line by placing a ‘\’ at the end of the first line
Invoking make

- Be sure that your description file:
  - is called makefile or Makefile
  - is in the directory with the source files
- `make` (builds the first target in the file)
- `make target(s)` (builds target(s))
- Important options:
  - `-n`: don’t run the commands, just list them
  - `-f file`: use file instead of [Mm]akefile
Basic Makefile example

program : main.o iodat.o dorun.o
  g++ -o program main.o iodat.o dorun.o
main.o : main.cc
  g++ -c main.cc
iodat.o : iodat.cc
  g++ -c iodat.cc
dorun.o : dorun.cc
  g++ -c dorun.cc
Simplifying the example Makefile with macros

OBJS = main.o iodat.o dorun.o
CC = /usr/bin/g++
program : ${OBJS}
  ${CC} -o $@ ${OBJS}
main.o : main.cc
  ${CC} -c $? 
iodat.o : iodat.cc
  ${CC} -c $? 
dorun.o : dorun.cc
  ${CC} -c $?
Suffix rules

- It’s still tedious to specifically tell `make` how to build each `.o` file from a `.c/,.cc` file.
- Suffix rules can be used to generalize such situations.
- A default suffix rule turns `.c/,.cc` files into `.o` files by running the command:
  ```
  ${CC} ${CFLAGS} -c $<
  ```
- `$<` refers to the prerequisite (file.cc)
Simplifying the example Makefile again

\[
\begin{align*}
\text{OBJS} & = \text{main.o iodat.o dorun.o} \\
\text{CC} & = \text{/usr/bin/g++} \\
\text{program : } & \{\text{OBJS}\} \\
& \quad \{\text{CC}\} \ -o \ @ \ \{\text{OBJS}\}
\end{align*}
\]
Other useful Makefile tips

- Include a way to clean up your mess

```makefile
clean:
    /bin/rm -f *.o core
```

- Include a target to build multiple programs

```makefile
all:
    program1 program2 program3
```