`gcc` and **`make`** primer

CS24 2011

(taken from CS11 C++
and Advanced C++ track)

Using **`gcc`**

- **`gcc`** has many arguments, but usually only use a few forms of invocation

```
gcc -Wall point.c lab1.c -o lab1
```

- **`-Wall`** causes compiler to report all errors
- **Always fix <u>all</u> compiler warnings!**
- If the compiler gives you a warning:
  - It is highly likely to either be a bug, or to cause bugs in the future
  - Often it's just reported as a warning because the language standard doesn't specify it to be an error

Using **`gcc`** (2)

- **`gcc`** has many arguments, but usually only use a few forms of invocation

```
gcc -Wall point.c lab1.c -o lab1
```

- **`-o`** ***`filename`*** specifies name of binary file
  - If unspecified, the default is **`a.out`**
- Can also list one or more source files on the command-line
  - Each source file is compiled separately
  - Then intermediate results are all linked together into a single executable binary

C Compilation

- You type:

```
gcc -Wall point.c lab1.c -o lab1
```

  - *What actually happens?*
- C compilation is a multi-step process:
  - Preprocessing
  - Compilation
  - Linking
- Different steps have different kinds of errors
  - …very helpful to understand what is going on!

C Compilation: Overview

- For the preprocessing and compilation phases, each source file handled separately!

```
gcc -Wall point.c lab1.c -o lab1
```

  - Compiler performs preprocessing and compilation on **`point.c`** and **`lab1.c`** separately
  - Produces **`point.o`** and **`lab1.o`**
    - (usually aren't saved to disk, unless compiler is told to do so)
- The linking phase combines the results of the compilation phase
  - **`point.o`** and **`lab1.o`** are combined into a single executable program, called **`lab1`**

The Preprocessor

- Step 1: The Preprocessor
  - Prepares source files for compilation
- Performs various text-processing operations on each source file:
  - Removes all comments from the source file
  - Handles preprocessor directives, such as **`#include`** and **`#define`**
- Example: lab1.c: **`#include "point.h"`**
  - Preprocessor *removes* this line from **`lab1.c`**, and *replaces* it with the contents of **`point.h`**

The Preprocessor (2)

- Another common example: C-style constants

```
#define MAX_WIDGETS 1000
```

- The preprocessor replaces all instances of **`MAX_WIDGETS`** with the specified text
  - After preprocessing, **`MAX_WIDGETS`** is gone, and source code contains 1000 instead
- For each input source file:
  - (e.g. **`lab1.c`**, **`point.c`**)
  - The preprocessor generates a translation unit, i.e. the input that the compiler actually translates into machine code

The Compiler

- The compiler takes a translation unit, and translates it from C code into machine code
  - i.e. from instructions that human beings understand, into instructions that your processor understands
- Result is called an <u>object file</u>
  - e.g. **`point.o`**, **`lab1.o`**
  - These are not runnable programs, but they contain machine-code instructions from your program

The Compiler: Object Files

- Object files are incomplete! They specify, among other things:
  - Each function that is <u>defined</u> within the translation unit, along with its machine code
    - e.g. **`point.o`** contains a definition of: **`double distance(Point *p1, Point *p2)`**
    - This includes the function's actual instructions!
  - Each function that is <u>referred to</u> by the translation unit, but whose definition is not specified
    - e.g. **`lab1.o`** uses the **`distance()`** function, but doesn't include a definition of the function

The Linker

- The linker takes the object files generated by the compiler, and combines them together
- Many object files refer to functions that they don't actually implement
  - Linker makes sure that every function is defined in *some* object file
- Two main kinds of errors:
  - Linker can't find the definition of a function
  - Linker finds multiple definitions of a function!

Linker Errors

- Example: you forget to include **`main()`**
  - Example output on Mac OS X:

```
Undefined symbols:
  "_main", referenced from:
      start in crt1.10.5.o
ld: symbol(s) not found
```

  - **`ld`** is the linker program for **`gcc`**
- These errors don’t occur during compilation
  - Compilation has succeeded, but the linker can’t find definitions for some functions

Final Compilation Notes

- Generally, compilers don't leave intermediate files around anymore
  - They use more efficient ways of passing translation units and object files to each other
- Can compile a source file without linking it:

```
gcc -Wall -c point.c
```

  - Performs preprocessing and compilation phases
  - Produces **`point.o`**
- Can also save outputs of preprocessor, compiler:

```
gcc -Wall --save-temps -c point.c
```

  - **`point.i`** is the result of running the preprocessor
  - **`point.s`** is a text version of the processor instructions

Makefiles

- The <u>makefile</u> describes build targets
  - Each target specifies its dependencies
  - Each target also specifies how to build that target from the dependencies
- Typical filename is **`Makefile`** or **`makefile`**
  - **`make`** looks for these by default
  - Preferred name is **`Makefile`**
  - Can specify another makefile using **`-f`** option

Makefiles (2)

- When **`make`** is run, a build target can be specified
    - **`make raytrace`**
  - If no target is specified, the first target in the makefile is run
    - **`make`**
  - First target is usually named **`all`**, and builds everything
  - Can also specify multiple build targets:
    - **`make clean raytrace`**
- Whitespace matters in makefiles!
  - Indentation is significant!
  - <u>Tabs</u> must be used for indentation!

Example Makefile

- Form of rules:

```
target : dependencies
        commands
```

- Example:

```
lab4 : main.o entry.o except.o config.o
        g++ -o lab4 main.o entry.o except.o config.o

main.o : main.cc entry.hh except.hh config.hh
        g++ -Wall -c main.cc

... (more rules)

clean :
        rm -f lab4 *.o *~
```

- Lines indented with tab characters – spaces won't work!
- A line can be continued on next line, by ending it with \
- A rule can specify multiple commands, if rules are separated by a blank line

Real Build Targets

- From our example:

```
main.o : main.cc entry.hh except.hh config.hh
        g++ -Wall -c main.cc
```

- In this case, **`main.o`** is a <u>real file</u>
- **`make`** will only build what is *needed*
  - If target file's date is older than any dependency, **`make`** will rebuild the target
- To force a file to be rebuilt, **`touch`** it:
    - **`touch main.cc`**
  - Sets file's modification-time to current system time
  - Touching a nonexistent file will create a new empty file

Phony Build Targets

- From our example:

```
clean :
        rm -f lab4 *.o *~
```

- In this case, **`clean`** is *not* a real file
- What if there happened to be a file named **`clean`** ?
  - Our rule wouldn't run!
  - **`make`** would see the "build-target" file, and assume it didn't have to run
- Use **`.PHONY`** to say that **`clean`** target isn't a file

```
.PHONY : clean
```

  - Now if a file named **`clean`** exists, **`make`** ignores it

Chains of Build Rules

- make figures out the graph of dependencies

```
lab4 : main.o entry.o except.o config.o
        g++ -o lab4 main.o entry.o except.o config.o
```

- If any of **`lab4`**'s dependencies don't exist, **`make`** will use their build rules to make them

```
main.o : main.cc entry.hh except.hh config.hh
        g++ -Wall -c main.cc
```

- **`make`** will give up if:
  - A dependency can't be found, and there's no build rule that shows how to make it

Makefile Variables

- Makefiles can define variables

```
OBJS = main.o entry.o except.o config.o
```

  – Can use variables in build rules

```
lab4 : $(OBJS)
        g++ $(OBJS) -o lab4
```

  – **$(*var-name*)** tells **make** to expand the variable

- Use variables to avoid listing the same things all over the place
  – Same as code reuse: only make changes in one place
- Makefile variable names are usually **ALL_CAPS**

Implicit Build Rules

- **`make`** already knows how to build certain targets
  - Those targets have built-in rules for building them
  - These built-in rules are called *implicit* build rules
- Example:
  - A makefile has **`main.o`** as a dependency, but no build rule
  - If **`main.c`** exists, **`make`** will use **`gcc`** to generate **`main.o`**
  - If **`main.cc`** exists, **`make`** will use **`g++`** to generate **`main.o`**
- **`make`** has quite a few implicit build rules
  - Read **`make`** documentation for more details!

Using Implicit Rules

- Implicit rules make your makefiles *much* shorter
  - Can leave out rules for all the object files

```
OBJS = main.o entry.o except.o config.o

lab4 : $(OBJS)
        g++ -o lab4 $(OBJS)

clean :
        rm -f lab4 *.o *~

.PHONY : clean
```

- What about header file dependencies?
  - Can specify rules for each object file:

```
main.o : entry.hh except.hh config.hh
```

    - No command – these rules just specify dependencies
  - **`makedepend`** auto-generates these from your source files!

Definitions of Implicit Rules

- Examples of implicit rules:

```
# C compilation implicit rule
%.o : %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@

# C++ compilation implicit rule
%.o : %.cc
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

- Variables are used for compiler and options!
  - **`CC`** is C compiler, **`CXX`** is C++ compiler
  - **`CFLAGS`**, **`CXXFLAGS`** are compiler-options
  - **`CPPFLAGS`** are the preprocessor flags
  - Default values are for **`gcc`** and **`g++`**
  - Can easily customize these rules, by setting these variables at the top of your makefile

Definitions of Implicit Rules

- Implicit rules use patterns:

```
# C compilation implicit rule
%.o : %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@

# C++ compilation implicit rule
%.o : %.cc
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

- Special syntax for pattern-matching
  - `%` matches the filename
  - **`$<`** is the first prerequisite in the dependency list
  - **`$@`** is the filename of the target
  - These **`$`**... values are called <u>automatic variables</u>
  - Other automatic variables too!
    - e.g. **`$^`** is list of *all* prerequisites in the dependency list

`make` References

- For more details, see the GNU **`make`** manual
  - http://www.gnu.org/software/make/manual/