CS11 Intro C++

FALL 2015-2016

LECTURE 5

C++ Compilation

- You type:

```
g++ -std=c++14 -Wall maze.cc testbase.cc \
    test-maze.cc -o test-maze
```

- What 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 preprocessing and compilation phases, each source file is handled separately

```
g++ -std=c++14 -Wall maze.cc testbase.cc \
    test-maze.cc -o test-maze
```

- Compiler performs preprocessing and compilation on each `.cc` file separately
  - Produces `maze.o`, `testbase.o`, `test-maze.o`
- The linking phase combines the results of the compilation phase
  - The three `.o` files are combined into a single executable program, called `test-maze`

The Preprocessor

- 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`
- `test-maze.cc` example: `#include "maze.hh"`
  - Preprocessor removes this line from `test-maze.cc`, and replaces it with the contents of `maze.hh`

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. `maze.cc`, `test-maze.cc`)
- The preprocessor generates a **translation unit**, i.e. the input that the compiler actually compiles
  - The translation unit contains only C++ source code, no preprocessor directives or comments

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 **object file**
  - e.g. `maze.o`, `testbase.o`, `test-maze.o`
- These are not runnable programs…
- They contain machine-code instructions from your program
  - e.g. `maze.o` contains the machine-code instructions that implement all operations in `maze.cc`

The Compiler: Object Files

- *Object files are incomplete!* They specify, among other things:
  - Each function that is defined within the translation unit, along with its machine code
  - e.g. `maze.o` contains a definition of `void Maze::clear()`
  - This includes the function's actual instructions!
- Each function that is referred to by the translation unit, but whose definition is not specified!
  - e.g. `test-maze.o` uses `Maze::clear()`, but it doesn't know the definition of this 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
  - The 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!

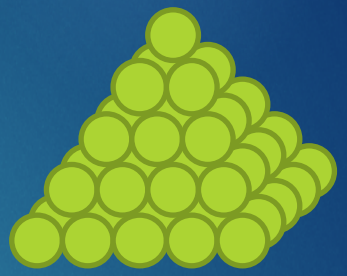
Linker Errors

- 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 `g++`
- These errors don't occur during compilation
  - Compilation has succeeded, but the linker can't find definitions for some functions
  - Therefore, this is not caused by e.g. a syntax error

More Linker Errors



- A file `pyramid.hh`:

```
int square(int x) {
  return x * x;
}

int pyramid_number(int n);
```

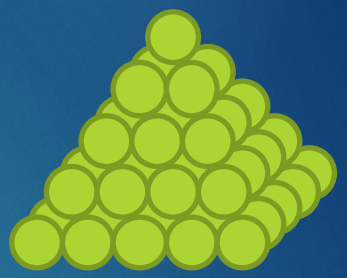
- Corresponding `pyramid.cc`:

```
#include pyramid.hh

int pyramid_number(int n) {
    int result = 0;
    for (int i = 1; i <= n; i++)
        result += square(i);
    return result;
}
```

- Symbols defined in `pyramid.o`:
  - `square()` function, `pyramid_number()` function

More Linker Errors (2)



- Now, `main.cc` prints a pyramid number:

```
#include <iostream>
#include "pyramid.hh"
using namespace std;
int main() {
    cout << "5th pyramid number is "
         << pyramid_number(5) << endl;
    return 0;
}
```

- Symbols defined in `main.o`:
  - `main()` function
  - Also `square()` function – definition was in `pyramid.hh`!

More Linker Errors (3)

- Now, both `main.o` and `pyramid.o` have a definition of `square()` function ☹
- Linker will complain:

```
g++ -Wall -std=c++14 pyramid.cc main.cc -o pyramid
duplicate symbol square(int) in:
    .../cculuqTZ.o
    .../cc6PQKcd.o
ld: 1 duplicate symbol for architecture x86_64
collect2: error: ld returned 1 exit status
```

- Again, not a syntax error!
  - Rather, a problem with the structure of the program

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:

```
g++ -std=c++14 -Wall -c maze.cc
```

  - Performs preprocessing and compilation
  - Produces `maze.o`
- Can save other output of preprocessor, compiler:

```
g++ -std=c++14 -Wall --save-temps -c maze.cc
```

  - `maze.ii` is result of running the preprocessor
  - `maze.s` is a text version of the processor instructions

Build Automation

- When a program grows beyond a certain size, compiling gets annoying…

```
g++ -std=c++14 -Wall maze.cc testbase.cc \
    test-maze.cc -o test-maze
```

  - Also, if only `maze.cc` is changed, why should `testbase.cc` and `test-maze.cc` be recompiled?
- Typical development process:
  - Write or modify some code
  - Compile
  - Test
  - Repeat until done…
- Automating this process saves lots of time and effort

make

- `make` is a standard tool for automating builds
  - Command-line utility, very ubiquitous!
  - Takes input files and produces output files, based on a "makefile"
  - Several versions of `make`: GNU, BSD, …
- `make` is often used for C and C++ projects
  - Sometimes other build tools are used for C/C++
  - CMake is becoming increasingly popular
  - Visual C++ provides `nmake` cmd-line build program
  - Other languages typically have their own build tools

Makefiles

- make requires a **makefile** that describes how to build your program
  - Typical filenames are `Makefile` (preferred) or `makefile`
  - Can specify a nonstandard makefile name with `make -f` *`some-other-makefile`*
- The makefile describes **build targets**
  - Files that need to be generated from other files
- Each target specifies its **dependencies** – the files needed to build the target
- Can also specify how to build the target from its dependencies

Example Makefile

- Example `Makefile`:

```
test-maze : test-maze.o maze.o testbase.o
        g++ -std=c++14 -Wall test-maze.o maze.o \
                testbase.o -o test-maze

maze.o : maze.cc maze.hh
        g++ -std=c++14 -Wall -c maze.cc

... (more rules for other .o files)

clean :
        rm -f test-maze *.o *~
```

- Lines indented with tab characters – spaces won't work!
- A line can be wrapped to next line by ending with \
- Can specify multiple commands in a rule, as long as rules are separated by blank lines

Running `make`

- When `make` is run, it automatically looks for the makefile in the current directory
- `make` will automatically try to build the first target specified in the makefile
- Usually, the first target in the makefile is named `all`, and it builds everything of interest

```
all : test-maze
```

  - (doesn't need to specify any commands)
- Can optionally specify one or more build targets:

```
make clean test-maze
```

Real Build Targets

- From our example makefile:

```
maze.o : maze.cc maze.hh
        g++ -std=c++14 -Wall -c maze.cc
```

- In this case, `maze.o` is a real file
- `make` will only build what is needed
  - If a target file's date is older than any dependency, `make` will rebuild that target
  - `make` will only rebuild the parts of the program that *actually changed*
- To force a file to be rebuilt, you can `touch` it

```
touch maze.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 test-maze *.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 do anything
- Use `.PHONY` to say that `clean` target isn't a file

```
.PHONY: clean
```

  - Now if a file named `clean` exists, `make` ignores it
  - (The `all` target is also phony…)

Chains of Build Rules

- make figures out the graph of dependencies

```
test-maze : test-maze.o maze.o testbase.o
        g++ -std=c++14 -Wall test-maze.o maze.o \
                testbase.o
```

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

```
maze.o : maze.cc maze.hh
    g++ -std=c++14 -Wall -c maze.cc
```

- `make` will give up if:
  - A dependency can't be found, and there's no build rule that shows how to make it
  - It finds a cycle in the graph of dependencies

Makefile Variables

- Makefiles can define variables

```
OBJS = maze.o testbase.o test-maze.o
```

- Can use variables in build rules

```
test-maze  : $(OBJS)
        g++ $(OBJS) -o test-maze
```

- `$(`*`var-name`*`)` tells `make` to expand the variable
  - Use variables to avoid listing the same things over and over again, all over the place
  - Same reasons as code reuse: state things once, so we only have to change things 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 `maze.o` as a dependency, but no corresponding build rule
  - If `main.c` exists, `make` uses `gcc` to generate `main.o`
  - If `main.cc` exists, `make` uses `g++` to generate `main.o`
- `make` has quite a few built-in implicit build rules!
  - Read `make` documentation for more details

Using Implicit Build Rules

- Implicit build rules make your makefiles much shorter

```
OBJS = test-maze.o maze.o testbase.o

all : test-maze

test-maze : $(OBJS)
        g++ -std=c++14 -Wall $(OBJS) \
                -o test-maze

clean :
        rm -f test-maze *.o *~

.PHONY: all clean
```

  - Can leave out rules for all the object files

Definitions of Implicit Rules

- Example definitions of implicit build 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++**

Leveraging Variables in Implicit Rules

- We want to use the implicit-rule variables in our makefiles!
- Example: specify `-Wall` and `-std=c++14` for compilation

```
CXXFLAGS = -Wall -std=c++14
OBJS = test-maze.o maze.o testbase.o

all : test-maze

test-maze : $(OBJS)
	$(CXX) $(CXXFLAGS) $(OBJS) -o test-maze \
		$(LDFLAGS)

clean :
	rm -f test-maze *.o *~

.PHONY : all clean
```

Definitions of Implicit Rules (2)

- Examples of implicit build rules:

```
# 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 `$...` values are called **automatic variables**
  - Other automatic variables too!
  - e.g. `$^` is list of all prerequisites in the dependency list

Using Automatic Variables

- Can use automatic variables to link our program

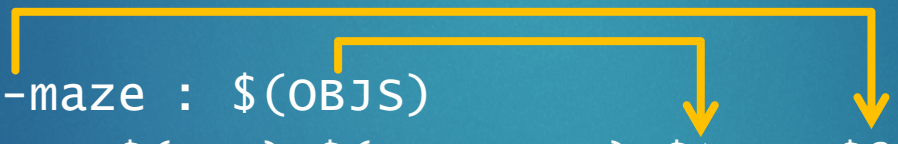
```
CXXFLAGS = -Wall -std=c++14
OBJS = test-maze.o maze.o testbase.o

all : test-maze

test-maze : $(OBJS)
        $(CXX) $(CXXFLAGS) $^ -o $@ $(LDFLAGS)

clean :
        rm -f test-maze *.o *~

.PHONY : all clean
```



make Reference

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

This Week's Assignment

- This week, you will write a maze generator!
- Get your `Maze` class passing all of its tests, if it isn't already
- Create a program `genmaze` that generates mazes!
  - Assignment will describe a couple of approaches for generating mazes
- Write a `Makefile` to build `test-maze` and `genmaze`, clean up temporary files, etc.

```
./genmaze 5 7
5 7
+---+---+---+---+---+---+---+
| S                 |       |
+---+---+---+   +   +---+   +
|       |       |   |       |
+---+   +   +---+   +---+---+
|       |   |               |
+   +---+   +---+---+---+   +
|       |           |       |
+   +   +---+---+   +   +---+
|   |                   E   |
+---+---+---+---+---+---+---+
```