CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2007
Lecture 16 – More about Makefiles and introduction to C++

Goal for Today

- A few more words about makefiles

- Starting C++

Building an Executable

- Last time, we saw the steps involved in building an executable
  - Preprocess and compile each `.c` file into a `.o` file
  - Optionally put `.o` files into a library (`.a` file)
  - Link everything together into an executable
- This process can get complicated for large systems
- Rebuilding a large system can also take a long time
- Therefore: need a way to manage the build process
  - We are studying one specific utility: make
  - But IDEs (Eclipse, VisualStudio) also do that for you

Make

- Two main goals
  - Automate the build process with a script
  - When a source file changes, rebuild only what is needed: keep track of *dependencies*

Recompilation Management

- The “theory” behind avoiding unnecessary compilation is a “dependency graph”
- To create target `t`, need
  - Sources $s_1, s_2, \ldots, s_n$ and a command `a`
- If `t` newers than all s_i, assume no reason to rebuild it
- Otherwise, recursive rebuild
  - If s_i is itself a target, check if need to rebuild it
  - If need to rebuild, use the given command a

Writing Makefiles

- Last time we saw the basics
- Today we will learn a few extra things to make our Makefiles simpler and more elegant
- Note: Make is language independent. Can have a makefile for Java, C, C++, latex documents, etc.

Variables

- You can define variables in a Makefile

```
OBJ = main-stack.o stack.o
main-stack: $(OBJ)
            gcc -o main-stack $(OBJ)
```

- Help avoid error-prone duplications
  - List of object files
  - List of executables
- In make, variables are often called macros

Default Macros

- There exists a lot of default macros
- You must respect the naming conventions
- Override defaults in the Makefile

```
CC = gcc
CFLAGS = -Wall -g
queue.o: queue.c queue.h
        $(CC) $(CFLAGS) -c queue.c
```

- Override defaults with environment variables

```
export CFLAGS =”-Wall -g”
```

- View list of macros: `make -p`

Revenge of Funny Characters

- Internal macros
  - `$@` designates the current target
  - `$^` designates all prerequisites
  - `$<` designates left-most prerequisite
- Pattern rules

```
%.o: %.c
	$(CC) $(CFLAGS) -c $<
```

- Basic ones already defined
  - They are called implicit rules

Dependencies

- Our Makefile is starting to look quite elegant
- But, we are still listing dependencies manually
  - Keeping track of dependencies is hard
  - It is easy to forget some header files
- This is not make's problem
  - Make has no understanding of programming languages. It only understands rules
- Because this is error-prone, there are often language-specific tools that can keep track of dependencies for you

Dependency-Generator Example

- `gcc -MM`
  - Useful variants include `-M` and `-MG` (`man gcc`)
  - Automatically creates a rule for you
  - One approach, run via a phony `depend` target

```
depend: $(SRC)
        $(CC) -M $^ > .depend
```

  - Then `include` the resulting file in your Makefile

```
include .depend
```

- `makedepend` combines many of these steps
- Read more if you are interested in this topic

Installing Program from Source

- You don't need to know this for the class
- Typical four steps when installing software

  `autoconf` (sometimes `setup` script instead)

  `configure --prefix=/where/to/install/`

  `make`

  `make install`

- Configure script: defines variables needed in the Makefile, performs various checks before compiling
- Configure script has many options so try

  `configure --help`

What You Need to Know

- Makefiles are a complicated topic
- For this class, you should be able to
  - Write Makefiles at the level of Makefile.v2
  - Read and understand Makefiles of the form Makefile.v3 and Makefile.v4
- For dependency generation (Makefile.v5), you only need to know that such a thing exists

Introduction to C++

- Object-oriented language like Java
- Based on C, manual memory management like in C
- Improves many features of C
  - C++ can be used solely as an "improved C" (without defining any classes)
- More complete standard library than C
- The "Standard Template Library" (STL)
  - A lot like Java "collections classes"
  - But not quite the same... so we will discuss them

Plan for This Week

- We will learn just enough C++ to get you started
- Today: the basics
  - Defining and using a simple class
  - Memory management
    - When objects are created and destroyed
    - Passing objects by value or by reference
- Wednesday: inheritance
- Friday: templates and STL

Hello World in C++

```
// Include header file from std library
// Note: “new style” header files have no .h
#include <iostream>

int main() {
   // Use standard output stream cout
   // and operator << to send “Hello World”
   // and an end line to stdout
   std::cout << "Hello World" << std::endl;
   return 0;
}
```

C++ Formatted Input/Output

- C++ I/O occurs in streams of bytes
- Stream insertion operator
  - Left shift operator (<<) designates stream output
  - Sends data from a variable to a stream
- Stream extraction operator
  - Right shift operator (>>) designates stream input
  - Extracts data from a stream into a variable
  - Example: `cin >> my_integer;`
- `cout`, `cin`, and `cerr` are stream objects
  - They are connected to `stdout`, `stdin` and `stderr`

Compiling C++ Programs

- It is standard for C files to have extension .c
- For C++, you can use: `.cpp`, `.cxx`, `.C`, **`.cc`**
- To compile C++ code, use g++ instead of gcc
- Standard example: "Hello World" (`hello.cc`)

```
g++ -Wall -o hello hello.cc
```

- Notes
  - In C++, there are no constraints on filenames
  - You can also put multiple classes in one file

Namespaces

```
#include <iostream>
using namespace std;

int main() {

  cout << "Hello World" << endl;
  return 0;

}
```

Namespaces

- A namespace allows us to group declarations under one name
- Namespaces help avoid name collisions and redefinition errors
- All the elements of the standard C++ library are declared within namespace `std`
- You should always use a namespace for your own declarations

Namespaces

```
#include <iostream>
using namespace std;
namespace MYSPACE {
  typedef struct {
    int a;
  } A;
}
int main() {
  MYSPACE::A sa;
  sa.a = 3;
  cout << sa.a << endl; // Prints: 3
  return 0;
}
```

Our First C++ Class

- Ok... now that we understand “Hello World”, we can get into the heart of things...

- We will examine a class called `Property`
  - We will point out differences between C++ and C
  - As well as difference between C++ and Java
- We will also discuss memory management

A Simple C++ Class

- Examine the `Property` class
  - Class definition in `.h` file
    - Includes member function declarations
    - Can also include function definitions (not recommended)
  - Member function definitions are in `.cc` file
  - Pay close attention to the constructor & destructor
  - Note the access specifiers: `public, private`
  - Note that we can use pointer `this` (in `toString`)
  - How the `static` attribute is declared and initialized
  - The use of namespaces

Member Access Specifiers

- They determine the type of access
  - public: accessible to everyone
  - private: accessible only to member functions
- The access specifiers can appear
  - In any order inside the header file
  - Multiple times, but preferably only once
- Default access mode is private

Function Overloading

- C++ enables function overloading where
  - Several functions have the same name
  - But different parameters
- The compiler selects the appropriate function
  - Matches arguments with parameters
- Examples:
  - The two: `adjustPrice` methods
  - The two constructors

Readings

- For more information, you can read one of many C++ tutorials
  - http://www.cplusplus.com/doc/tutorial/