HACK YALE

RAPID FIRE C++

WWW.HACKYALE.COM

HACK YALE

RAPID FIRE C++

C++

AN INTRODUCTION

COURSE INFORMATION

LOGISTICS, YAY!

HACK YALE

HI, EVERYONE

- I'm Alex Reinking, TC '16.
- I'm a Math/CS double-major.
- I've been coding for 13 years, in C++ for 10.
- I'm a peer tutor for 223/323, so we might have met before.

ABOUT THIS CLASS

- I assume you've all worked in a procedural language before.
  - C, Java, Python, Ruby, JavaScript, etc. - all OK
  - If you've only taken 201 - see me after
- We meet once per week. The time and place should stay the same.
- We will cover C++ *very quickly*. I won't be grading homework, but I will give out exercises.

WHY SHOULD I LEARN C++?

WHAT IS IT GOOD FOR?

HACK YALE

C++ IS EVERYWHERE!

- C++ is the leading programming languages in many industries because it is fast, reliable, and has a wealth of code written for it already
- It isn't resource-intensive (unlike Java), and can run on pretty much any system.
- It is used at major companies such as Google, Microsoft, and Adobe.

LANGUAGE USAGE BY TIOBE INDEX

- The five most popular programming languages:
  - C
  - Java
  - Objective C
  - C++
  - C#
- C++ is the fourth-most used, but since C++ is a superset of C, by knowing one, you know the other.

INDUSTRIES THAT LOVE C++

- Game Development (XBox, PlayStation, etc.)
- Finance (High Speed Algorithmic Trading)
- Scientific and High-Performance Computing
- Application Software
  - Google Chrome
  - Microsoft Office
  - Adobe Photoshop

THE C++ ECOSYSTEM

THE BEST TOOLS TO USE

HACK YALE

ABOUT C++

- C++ is a *compiled* language.
  - C, Haskell, Objective C, and Swift are compiled, too
  - Python, Ruby, and PHP are *interpreted*
    - Java and C# are in a weird place
- You must send your code through a compiler in order to actually run it
- So how do you compile C++?

COMPILERS

- On Windows, there are two main options
  - Microsoft Visual C++ (MSVC)
  - MinGW (Minimalist GNU for Windows)
- On OS X, Apple provides Clang by default
- On Linux, everyone uses G++.
  - `sudo apt-get install g++`
- If you're rich, you can use the Intel C++ Composer.

COMPILERS

- They all perform the same task, but do so with varying degrees of capability.
- MSVC is for Windows development only.
- Clang and G++ are roughly equivalent.
- The Intel C Compiler produces insanely fast code.
- All compilers include a *linker*, which combines all the source files you compile.

EDITORS

- You can always use Sublime Text
- I prefer Qt Creator
  - http://www.qt-project.org
  - Includes a compiler, or uses your system one
- Modern IDEs offer significant advantages in code completion, type awareness, automatic refactoring…
- All you Java folks, look up IntelliJ IDEA.

LIBRARIES

- Code reuse is an important part of programming.
- Collections of general components are called libraries
- In C++, there are a lot of good ones
  - STL - built in. Has all the basic data structures and algorithms.
  - Boost - includes everything the STL doesn't have. Advanced data structures, monadic parsing, crazy stuff. Often becomes standard.
  - Qt - App development: GUI, Databases, Networking, and more

CODE ORGANIZATION

WHERE DOES IT ALL GO?

HACK YALE

SOURCES AND HEADERS

- In C++, as in many other languages, there are:
  - Declarations: The name and type of a variable or function
  - Definitions: The value or body of a variable or function
- We split these into ***header*** (.h) and ***source*** (.cpp) files, respectively. Source files include header files (which can include other headers).
- Once a source has access to a declaration, the compiler finds the definition at the end.

SOURCES AND HEADERS

- Only sources are compiled directly. The headers they include get pasted in and are compiled, too.
- Because headers include others, we need to prevent loops. There are two ways to do this:
  - `#pragma once`
  - ```
    #ifndef INCLUDE_GUARD
    #define INCLUDE_GUARD
    // declarations go here...
    #endif
    ```
    ```

MORE CONVENTIONS

- Like in Java, when we define classes in C++, we keep each one separate. Except instead of one file, we go in header/source pairs.
- The class *declaration* goes in the header, while the *definitions* (the bodies of the functions) go in the source.
- Then, when a different source needs to use a class, it can just include the header. The linker will find the definitions later.

MORE CONVENTIONS

- When you have large projects, you'll want to split your source files into directories to keep things neat.
- In the top-level directory, you'll keep a few folders for things like the source code, the test code, the compiled files, image assets, data, etc.
- A good directory layout looks like other directory layouts.

EXAMPLE DIRECTORY STRUCTURE

- From the top level project directory, you might have:
  - src/ - all the source code
    - main.cpp - the entry point for your program
    - module1/ - the code for module 1
  - tests/ - all the source code for automatic tests
  - build/ - all the compiled (object) files
  - Makefile - (coming up next)

CODE STYLE

- Almost anything works, but be consistent.
- Short lines, short functions.
- Curly braces in the same place every time.
- Regular indentation
- Few blank lines
- etc.

BUILD SYSTEM BASICS

HOW C++ IS ASSEMBLED

HACK YALE

BASIC CONCEPTS

- Compilation Unit
  - One source file maps to one compilation unit.
  - Everything compiles, and type checks, but not all of the definitions are available.
- Compilation units are *linked* together to make
  - An executable: a full process that can be run
  - A shared library: fully compiled code that can be linked into other executables/libraries.

BASIC CONCEPTS

- All definitions must be available at link-time
  - Otherwise, you'll get a linker error
- This applies to your code as well as that of 3rd party libraries
- So, we get two phases:
  - Compilation: All of the source (.cpp) files are compiled (.o)
  - Linking: All of the objects (.o) are linked together into the artifact (.exe, .dll, .so, .dylib, etc)

THE BUILD PROCESS

FROM CODE TO EXECUTION

MAKEFILES!

- GNU Make
  - Simple, straight-forward build tool.
  - Makes sure everything gets built in the right order
    - Boils down to command-line commands you could write yourself
- Standard on all UNIX systems
- Complemented by the (very-complicated) autotools.

EXAMPLE MAKEFILE

```
# Note: This is overly simple
myProgram: build/module.o build/main.o
	g++ $^ -o $@  # need an actual tab in front

build/module.o: src/module1/module.cpp
	g++ $^ -c -o $@

build/main.o: src/main.cpp
	g++ $^ -c -o $@  # notice the duplication?
```

MAKEFILE ISSUES

- Makefiles quickly become unwieldy for large projects
- Make doesn't handle nested directories very well
- Solutions?
  - Use a flat directory structure (put all sources in /src, all headers in /include)
  - Use an IDE to manage the build (MSVC, Xcode)
  - Generate a complicated Makefile automatically (QMake, CMake)

IDE BUILDS

- Advantages:
  - Fast, easy to use. No need to think about it.
  - Perfect integration into the IDE you're using
- Disadvantages:
  - Tend only to work for you.
  - Need the IDE installed.
  - Hard to distribute code

MAKEFILE GENERATION

- GNU Autotools
  - Horribly complicated, based on shell scripts.
- QMake
  - Straightforward generation of Makefiles from a list of sources
- CMake
  - Like QMake, but more flexible. Can generate Makefiles and IDE projects.

QMAKE EXAMPLE

```
# Filename: myProgram.pro
SOURCES += src/main.cpp \
           src/module1/module.cpp
HEADERS += src/module1/module.h
```

HOW TO BUILD WITH QMAKE?

- Simple at the command line:
  - `qmake myProject.pro`
    `make`
    - That's it!
- QMake can actually generate its own files for simple projects!
  - `qmake -project`

OUR FIRST C++ APP

THIS ONE IS SIMPLE