CS11 Intro C++
Spring 2018 – Lecture 1
Welcome to CS11 Intro C++!

• An introduction to the C++ programming language and tools
• Prerequisites:
  • CS11 C track, or equivalent experience with a curly-brace language, is encouraged but not required
• No books are required for this course
  • Lecture slides and assignments are sufficient
  • Lecture recordings will also be available
• If you want some reference books:
  • A Tour of C++, 2nd Edition
    • An overview and survey of C++, by its creator
    • Contains good advice on proper C++ usage and recommended idioms
    • Better for more experienced programmers
Welcome to CS11 Intro C++! (2)

• An introduction to the C++ programming language and tools

• Prerequisites:
  • CS11 C track, or equivalent experience with a curly-brace language, is encouraged but not required

• No books are required for this course
  • Lecture slides and assignments are sufficient
  • Lecture recordings will also be available

• If you want some reference books:
  • Programming, 2
    • Learning to program, using C++, by its creator
    • A much expanded version of the previous book
    • Good for novice programmers
Assignments and Grading

- Each lecture has a corresponding assignment for exploring the material
- Labs are due approximately one week later, at noon
  - e.g. this term labs will be due on Fridays at noon
  - Submit on csman
- Labs are given a 0..3 grade, meaning:
  - 3 = excellent (masters all important parts)
  - 2 = good (demonstrates mastery of key idea; a few minor issues)
  - 1 = insufficient (not passing quality; significant bugs that must be addressed)
  - 0 = incorrect (worthy of no credit)
- Must receive at least 75% of all possible points to pass the track
- Can submit up to 2 reworks of assignments to improve grade
- Not uncommon for initial submission to get a 0!
  - Don’t take it personally; it’s really not a big deal in CS11 tracks
C++ Compilers

• Two main C++ compilers in use these days
  • GNU g++
    • Most widely used on Linux systems
    • Typically used in cygwin on Windows systems
  • LLVM clang++
    • The default compiler on Apple MacOSX
    • clang++ emulates some basic g++ functionality, but also leaves out many options
• If unsure, you can find out what you are using:
  • “g++ --version” outputs the compiler version
• Example output on a Mac:
  • LLVM: “Apple LLVM version 6.0 (clang-600.0.57)”
  • GNU: “g++ (MacPorts gcc49 4.9.3_0) 4.9.3”
C++ Compilers (2)

• As long as we can compile and run your code with either GNU g++ or LLVM clang++, you’re fine
• Can specify the version of C++ to use
  • g++ -std=c++14 ...
  • clang++ -std=c++14 ...
  • (or use -std=c++11 if your compiler doesn’t support C++14)
• Most annoying difference between g++ and clang++ is that the debuggers are very different
  • g++ provides gdb
  • clang++ provides lldb
  • The debugger commands are significantly different
• If you are on a Mac and want to use g++/gdb, use Homebrew or MacPorts to install them
  • Make sure your path is set up to find GNU g++, and not clang’s “fake g++”
C++ Origins

- Original designer: Bjarne Stroustrup, AT&T Bell Labs
- First versions called “C with Classes” – 1979
  - Most language concepts taken from C
    - “C with Classes” code was translated into C code, then compiled with the C compiler
    - Class system conceptually derived from Simula67
- Name changed to “C++” in 1983
- Continuous evolution of language features
  - (as usual)
    - Renewed development recently, with C++11, C++14 and upcoming C++17 standard updates
C++ Philosophy

“Close to the problem to be solved”
• Elegant, powerful abstractions
• Strong focus on modularity

“Close to the machine”
• Retains C’s focus on performance, and ability to manipulate hardware and data at a low level
  • Good language e.g. for games programming, systems programming, etc.
• “You don’t pay for what you don’t use.”
  • Some features have additional cost (e.g. classes, exceptions, runtime type information)
  • If you don’t use them, you don’t incur the cost
C++ Components

C++ Core Language
• Syntax, data types, variables, flow control, ...
• Functions, classes, templates, ...

C++ Standard Library
• Many useful classes and functions written using the core language
• Generic strings, IO streams, exceptions
• Generic containers and algorithms
  • The Standard Template Library (STL)
• Multithreading support
• Several other useful facilities
Example C++ Program

- Hello, world!

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!\n";
    return 0;
}
```

- **main()** function is program’s entry point
  - Every C++ program must have exactly one `main()` function
- Returns 0 to indicate successful completion, nonzero (typically 1..63) to indicate that an error occurred
Compilation

• Save your program in hello.cpp
  • Typical C++ extensions are .cpp, .cc, .cxx
  • Typical C++ header files are .h, .hpp, .hh, .hxx

• Compile your C++ program
  > g++ -std=c++14 -wall hello.cpp -o hello
  > ./hello
  Hello, world!

• Typical arguments:
  • –wall Reports all compiler warnings. **Always fix these!!!**
  • –o file Specifies filename output by the compiler
    • Defaults to a .out, which isn’t very useful...
Console IO in C++

• C uses `printf()`, `scanf()`, etc.
  • Defined in the C standard header `stdio.h`
  • `#include <stdio.h>` (or `<cstdio>` in C++)
• C++ introduces “Stream IO”
  • Defined in the C++ standard header `iostream`
  • `#include <iostream>`
• In this class, we will use C++ stream IO
  • `printf/scanf` can be useful in C++ programs, but we are here to learn C++!

• `cin` – console input, from “stdin”
• `cout` – console output, to “stdout”
• Also `cerr`, which is “stdout,” for error-reporting.
Stream Output

• The `<<` operator is **overloaded** for stream-output
  • Compiler figures out when you mean “shift left” and when you mean “output to stream,” from the context
  • Supports all primitive types and some standard classes, e.g. C++ strings

• Example:
  ```cpp
  string name = "series";
  int n = 15;
  double sum = 35.2;
  cout << "name = " << name << "\n"
       << "n = " << n << "\n"
       << "sum = " << sum << "\n";
  ```

• **Note:** Line up `<<` operators to improve code readability
Stream Input

• The >> operator is overloaded for stream-input
  • Also supports primitive types and C++ strings.

• Example:
  
  ```
  float x, y;
  cout << "Enter x and y coordinates: ";
  cin >> x >> y;
  ```

• Input values are whitespace-delimited.
  
  ```
  Enter x and y coordinates: 3.2 -5.6
  Enter x and y coordinates: 4 35
  ```
C++ Namespaces

• **Namespaces** are used to group related items
• All C++ Standard Library code is in the `std` namespace
  • `string`, `cin`, `cout` are part of Standard Library
• Can either write `namespace::name` everywhere...
  ```cpp
  std::string name;
  std::cin >> name;
  std::cout << "Hello, " << name << "\n";
  ```
• Or, declare that you are using the namespace!
  ```cpp
  using namespace std;
  string name;
  cin >> name;
  cout << "Hello, " << name << "\n";
  ```
• `namespace::name` form is called a **qualified name**
C++ Classes

- C++ classes are made up of **members**

- **Data members** are variables that appear in objects of the class’ type
  - They store the object’s state
  - Also called **member variables** or **fields**

- **Member functions** are operations that can be performed on objects of the class’ type
  - These functions usually involve the data members

- Several different categories of member functions
Member Function Types

• **Constructors** initialize new instances of a class
  • Can take arguments, but not required. No return value.
  • Every class has at least one constructor
  • No-argument constructor is called **default constructor**
  • Several other special kinds of constructors too

• **Destructors** clean up an instance of a class
  • This is where an instance’s *dynamically-allocated* resources are released
    • (The compiler knows how to clean up everything else)
  • No arguments, no return value
  • Every class has **exactly one** destructor
Member Function Types

- **Accessors** allow internal state to be retrieved
  - Provide control over when and how data is exposed
- **Mutators** allow internal state to be modified
  - Provide control over when and how changes can be made

- Accessors and mutators guard access to (and mutation of) an object’s internal state values
- Generally don’t want to expose internal state!
  - Instead, provide accessors and mutators to govern when and how internal state is exposed and manipulated
Abstraction and Encapsulation

• **Abstraction:**
  • Present a clean, simplified interface
  • Hide unnecessary detail from users of the class (e.g. implementation details)
  • They usually don’t care about these details!
  • Let them concentrate on the problem they are solving.

• **Encapsulation:**
  • Allow an object to protect its internal state from external access and modification
  • The object itself governs all internal state-changes
  • Methods can ensure only valid state changes
Declarations and Definitions

• C++ distinguishes between the declaration of a class, and its definition.

• The **declaration** describes member variables and functions, and their access constraints.
  • This is put in the “header” file, e.g. `point.h`

• The **definition** specifies the behavior – the actual code of the member functions.
  • This is put in a corresponding `.cpp` file, e.g. `point.cpp`

• Users of our classes include only the declarations
  • `#include "point.h"
  • People usually don’t care how the types work internally; just how to use them to solve other problems
C++ Access Modifiers

• The class declaration states what is exposed and what is hidden.

• Three access-modifiers in C++
  • public – Anybody can access it
  • private – Only the class itself can access it
  • protected – We’ll get to this later...

• The default access-level for classes is private.

• In general, other code can only access the public parts of your classes.
// A 2D point class
class Point {
    double x, y;  // Data-members

public:
    Point();  // Constructors
    Point(double x, double y);

    ~Point();  // Destructor

    double get_x();  // Accessors
    double get_y();
    void set_x(double x);  // Mutators
    void set_y(double y);
};
#include "point.h"

// Default (aka no-argument) constructor
Point::Point() {
    x = 0;
    y = 0;
}

// Two-argument constructor - sets point to (x, y)
Point::Point(double x, double y) {
    this->x = x;
    this->y = y;
}

// Cleans up a Point object.
Point::~Point() {
    // No dynamically allocated resources; nothing to do!
}
Variable Shadowing

• A somewhat confusing situation:
  ```cpp
  Point::Point() {
    x = 0;
    y = 0;
  }
  
  Point::Point(double x, double y) {
    this->x = x;
    this->y = y;
  }
  ```

• In C++, variables in an inner scope can *shadow* a variable in an outer scope
  • The data-members x and y are defined at the object scope
  • Additionally, function arguments x and y are arguments to the constructor, and these shadow the data-members

• Consequence: If you say “x” or “y” by itself, compiler assumes you mean the function argument, not the data-member
  • (In general, compiler uses the variable at the narrowest scope)
Variable Shadowing (2)

- A somewhat confusing situation:
  ```cpp
  Point::Point() {
      x = 0;
      y = 0;
  }

  Point::Point(double x, double y) {
      this->x = x;
      this->y = y;
  }
  ```

- A simple solution: use `this` to resolve the ambiguity, when needed
  - “this” is a pointer to the object that member function is being invoked on
  - Built into the C++ language, available in `member-functions`, but not regular functions (exactly like Java “this” or Python “self”)
  - In this example, “this” has the type `Point*`, because the member function is part of the Point class.
// Returns X-coordinate of a Point
double Point::get_x() {
    return x;
}

// Returns Y-coordinate of a Point
double Point::get_y() {
    return y;
}

// Sets X-coordinate of a Point
void Point::set_x(double x) {
    this->x = x;
}

// Sets Y-coordinate of a Point
void Point::set_y(double y) {
    this->y = y;
}
Using the **Point** Type

- Now we have a new type to use!
  ```cpp
  #include "point.h"
  Point p1;           // Calls default constructor
  Point p2{3, 5};    // Calls 2-arg constructor
  cout << "P2 = (" << p2.get_x()
  << "," << p2.get_y() << ")\n";
  p1.set_x(210);
  p1.set_y(154);
  ```

- Point’s private members cannot be accessed directly.
  - `p1.x = 452;` // Compiler reports an error!
  - `cout << p2.y;` // Compiler reports an error!
The C++ `std::string` Class

• C++ retains the C notion of `char*` as a “string”
  • An array of `char` values, terminated with a 0 value (a.k.a. “the null character” or “NUL”)
• Typically difficult / bug-prone to manipulate in complex ways...
  • Have to manually allocate and reallocate space to hold string data
  • Can easily write past end of string (buffer overflows, exploits!)
  • Can easily forget to free memory used by C strings

• C++ also introduces a new `std::string` type
  • Resizable string that keeps data in heap memory
  • `#include <string>`

• Provides many features over `char*` strings
  • Can manipulate strings easily, without manual memory management
  • Supports stream IO with `>>` and `<<` operators

• Prefer `string` to `char*`, wherever possible!!!
The C++ `std::string` Class (2)

- Usage of `std::string` is very intuitive
  ```cpp
  string name;
  cout << "What is your name? ";
  cin >> name;
  cout << "Hello " << name << "!\n";
  ```
- Setting initial values, or mutating string values, is also easy
  ```cpp
  string favorite_color{"green"};
  string mood = "happy";
  mood = "cheery";
  ```
- Will cover C++ string functionality in much more detail in the future!
This Week’s Homework

• For the next few weeks, we will build a simple units-conversion utility
• When finished, it will be quite powerful

• This week:
  • Start practicing the basic concepts of C++ class declaration, and start creating the machinery for our utility
  • Focus on good coding style and commenting
  • Figure out what C++ compiler you have, and how to invoke it
  • Figure out how to compile your program on your computer
  • Test your program’s correctness