CS11 Intro C++

FALL 2015-2016

LECTURE 1

Welcome!

- Introduction to C++
  - Assumes general familiarity with C syntax and semantics
  - Loops, functions, pointers, memory allocation, structs, etc.
- I am revising the track this term…
  - Update track to include C++11 and C++14 features
  - Update labs to be more fun and challenging
  - The track may be a bit rough this term ☺

Lectures and Assignments

- Aiming for 7-8 lectures
  - Lectures will be posted on CS11 course website
- Aiming for 7-8 lab assignments
  - Assignments will cover a variety of C++ topics
- Both lectures and assignments are available at http://courses.cms.caltech.edu/cs11/
- Intro C++ track requires a CS cluster account
  - Assignments are submitted on csman
  - https://csman.cms.caltech.edu
  - csman uses CS cluster for user authentication

Submissions

- You can also write your submissions on the CS cluster, but you aren't required to
- As long as we can compile and run your code with GNU g++ or LLVM clang, you're fine
  - `g++ -std=c++14 ...`
  - `clang -std=c++14 ...`
  - (or use `-std=c++11` if your compiler doesn't support `c++14`)
- Submissions will be given a grade of 0..3
  - 3 = perfect; 0 = major issues that need resolved
- **Submissions are due on Wednesdays at noon**
  - -0.5 points per day will be assessed for late work

C++ Compilers

- 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 it also leaves out a lot of options
- You can find out which version you are using:
  - "`g++ -version`" outputs the compiler version
- Example output:
  - LLVM: "Apple LLVM version 6.0 (clang-600.0.57)"
  - GNU: "g++ (MacPorts gcc49 4.9.3_0) 4.9.3"

C++ Compilers (2)

- 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
  - "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
  - A collection of useful classes and functions written using the core language
  - Generic strings, IO streams, exceptions
  - Generic containers and algorithms
    - The Standard Template Library (STL)
  - Several other useful facilities

Example C++ Program

- Hello, world!

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!" << endl;
    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.cc`
  - Typical C++ extensions are `.cc`, `.cpp`, `.cxx`
  - Typical C++ header files are `.hh`, `.hpp`, `.hxx`
- Compile your C++ program

```
> g++ -Wall hello.cc -o hello
> ./hello
Hello, world!
```

- Typical arguments:
  - `-Wall` Reports all compiler warnings
    - **Always fix these!!!**
  - `-o` *`file`* Specifies output filename
    - 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>`

- `cin` – console input, from "stdin"
- `cout` – console output, to "stdout"
- Also `cerr`, which is "stderr," 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
  - `endl` means "end of line" in C++
- Example:

```
string name = "series";
int n = 15;
double sum = 35.2;
cout << "name = " << name << endl
     << "n = " << n << endl
     << "sum = " << sum << endl;
```

Stream Input

- The >> operator is overloaded for stream-input
  - Also supports primitive types and 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++ Stream IO Tips

- Generally a bad idea to mix C-style IO and C++ stream IO in the same program
  - Both use the same underlying OS resources
  - Either API can leave stream in a state unexpected by the other one
- Sometimes it's very helpful to use `printf()` and `scanf()` in C++ programs
  - Much easier to format output, etc.
- In this class, use C++ IO in your programs ☺
- Can use `endl` to end lines, or `"\n"`.
  - These are actually not the same in C++
  - Use `endl` in this class

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
- Either write `namespace::name` everywhere…

```
std::cout << "Hello, world!" << std::endl;
```

- Or, declare that you are using the namespace!

```
using namespace std;
...
cout << "Hello, world!" << endl;
```

- `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

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

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.

Declarations and Definitions

- C++ makes a distinction 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.hh`
- The **definition** specifies the behavior – the actual code of the member functions.
  - This is put in a corresponding `.cc` file, e.g. `Point.cc`
- Users of our class include the declarations
  - `#include "Point.hh"`

Point Class Declaration – point.hh


```
// A 2D point class!
class Point {
    double x_coord, y_coord;     // Data-members

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

    ~Point();                    // Destructor

    double getX();               // Accessors
    double getY();
    void setX(double x);         // Mutators
    void setY(double y);
};
```

Defining the Point's Behavior – point.cc (1)


```
#include "Point.hh"

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

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

// Cleans up a Point instance.
Point::~Point() {
    // no dynamically allocated resources; nothing to do!
}
```

Defining the Point's Behavior – point.cc (2)


```
// Returns X-coordinate of a Point
double Point::getX() {
    return x_coord;
}

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

// Sets X-coordinate of a Point
void Point::setX(double x) {
    x_coord = x;
}

// Sets Y-coordinate of a Point
void Point::setY(double y) {
    y_coord = y;
}
```

Using the `Point` Type

- Now we have a new type to use!

```
#include "Point.hh"
...
Point p1;               // Calls default constructor
Point p2(3, 5);         // Calls 2-arg constructor
cout << "P2 = (" << p2.getX()
     << "," << p2.getY() << ")" << endl;
p1.setX(210);
p1.setY(154);
```

- Point's private members can't be accessed directly.
  - `p1.x_coord = 452;   // Compiler reports an error.`

Local Variables and Default Constructors

- When initializing an object local variable using default constructor: **Don't use parentheses!**

```
Point p1;
```

- If you use parentheses, C++ will think you are declaring a function:

```
Point p1();
```

  - Compiler assumes this is a function (or function-pointer) named p1, that takes no arguments and returns a Point object

- Other situations may allow parentheses with the default constructor, e.g.

```
p1 = Point();
outputPoint(Point());
```

C++ Function Arguments

- Function arguments in C++ are passed **by-value**
  - A copy of each argument is made
  - The function works with the copy, not the original
- Example:

```
void outputPoint(Point p) {
    cout << "(" << p.getX()
         << "," << p.getY() << ")";
}
...
Point loc(35,-117);
outputPoint(loc);     // loc is copied
```

- Copying lots of objects can get very expensive…

C++ References

- C++ introduces **references**
  - A reference is like an alias for a specific variable
  - Using the reference is exactly like using what it refers to
- Updating our function to pass its argument **by-reference**:

```
void outputPoint(Point &p) {
  cout << "(" << p.getX()
       << "," << p.getY() << ")";
}
```

  - `p` is of type `Point&` – "reference to a `Point` object"
- Now, `outputPoint()` operates on the actual `loc` object

```
Point loc(35,-117);
outputPoint(loc);  // loc is passed "by-reference"
```

C++ References (2)

- The referenced variable can be changed through the reference

```
// Rotate a point 90 degrees counter-clockwise
void rotate90ccw(Point &p) {
    double x = p.x;
    p.x = -p.y;
    p.y = x;
}
...
Point p(5, 3);
rotate90ccw(p);  // Changes p to (-3, 5)
```

- Clearly wouldn't work with pass-by-value semantics!

C++ References (3)

- C++ references must refer to a variable through their entire lifetime

```
void bad() {
    Point &p; // Error:  p must be initialized
}
```

  - Not possible to set C++ references to a "null" value
- Also, it's not possible to change which variable a reference refers to, during its lifetime

```
Point p1(3, 5), p2(4, 6);
Point &p = p1;
p = p2;
```

  - Doesn't change `p` to reference `p2` !
  - Rather, assigns `p2`'s values to `p1` (`p` is an alias for `p1`)

C++ Reference Guidelines

- When writing functions, always pass object arguments by-reference, not by value
  - YES: `void outputPoint(Point &p)`
  - NO: `void outputPoint(Point p)`
- Don't bother to pass primitive values by-reference
  - YES: `int max(int a, int b)`
  - NO: `int max(int &a, int &b)`
  - It will almost certainly make your code slower
- When passing objects by-reference, be careful to avoid unintended side-effects
  - A buggy function could accidentally change an object passed by-reference… difficult to track down…
  - (We will talk about how to prevent this in the future.)

Spacing Out

- These are all equivalent:

```
int *p;      // Space before *
int* p;      // Space after *
int * p;     // Space before and after *
```

- Same with references:

```
Point &p;    // Space before &
Point& p;    // Space after &
Point & p;   // Space before and after &
```

- Best practice: space before, no space after
  - Example: `int* p, q;`
  - What are the types of `p` and `q`?
  - `p` is an `int*`
  - `q` is an `int`, *not* an `int*`
  - The * is associated with the variable, not the type-name

This Week's Homework

- Create a simple 3D point class in C++
  - Add a few member functions to the class
- Use your class in a simple math program
- Use console IO to drive your program

- Learn how to compile and run your program

- Test your program to make sure it's correct