CS11 – Introduction to C++

Winter 2011-2012

Lecture 1

Welcome!

- Introduction to C++
  - Assumes general familiarity with C syntax and semantics
    - Loops, functions, pointers, memory allocation, structs, etc.
- 8 Lectures (~1 hour)
  - Slides posted on CS11 website
  - http://courses.cms.caltech.edu/cs11
- No textbook is required
  - All necessary material covered in slides, or available online
- 7 Lab Assignments – on course website
  - Usually available on Monday evenings
  - Due one week later, on Monday at 12:00 noon

Assignments and Grading

- Labs focus on lecture topics
  - …and lectures cover tricky points in labs
  - Come to class! I give extra hints. ☺
- Labs are given a score in range 0..3, and feedback
  - If your code is broken, you will have to fix it.
  - If your code is sloppy, you will have to clean it up.
- Must have a total score of 18/24 to pass CS11 C++
  - (Lab 4 is a two-week lab, and is worth 6 points.)
  - Can definitely pass without completing all labs
- Please turn in assignments on time
  - You will lose 0.5 points per day on late assignments

Lab Submissions

- Using csman homework submission website:
  - https://csman.cs.caltech.edu
  - Many useful features, such as email notifications
- Must have a CS cluster account to submit
  - csman authenticates against CS cluster account
- CS cluster account also great for doing labs!
  - Can easily do the labs on your own machine, as long as your work builds with a recent g++ version

“Tips and Tricks” Books

- Many great books!
  - Effective C++, More Effective C++
    - Scott Myers
  - Exceptional C++, More Exceptional C++
    - Herb Sutter
  - Exceptional C++ Style
    - Herb Sutter
- These books teach you how to use C++ *well*
  - Not necessary for this track
  - A *great* investment if you expect to use C++ a lot

C++ Origins

- Original designer: Bjarne Stroustrup
  - AT&T Bell Labs
- First versions called “C with Classes” – 1979
  - Most language concepts taken from C
  - Class system conceptually derived from Simula67
- Name changed to “C++” in 1983
- Continuous evolution of language features
  - Many enhancements to class system; operator overloads; references; const; templates; exceptions; namespaces; …

C++ Philosophy

- "Close to the problem to be solved"
  - Ability to build elegant and powerful abstractions
  - Strong focus on modularity
  - Big enhancements to C type-system
- "Close to the machine"
  - Retains C's focus on performance
  - Also retains C's ability to do low-level manipulation of hardware and data

Two Components of C++

- The C++ core language
  - Syntax, data-types, variables, flow-control, …
  - Functions, classes, templates, …
- The C++ Standard Library
  - A collection of useful classes and functions written in the core language
  - Generic strings, streams, exceptions
  - Generic containers and algorithms
    - The Standard Template Library (STL)

My First 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 contain exactly one **`main()`** function

Make It Go.

- Save your program in **`hello.cc`**
  - Typical C++ extensions are **`.cc`**, **`.cpp`**, **`.cxx`**
- Compile your C++ program

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

- We are using GNU C++ compiler, **`g++`**
  - Several other C++ compilers too, but **`g++`** is widely available and widely used

Console IO in C++

- C uses **`printf()`**, **`scanf()`**, etc.
  - Defined in the C standard header **`stdio.h`**

    `#include <stdio.h>`
- 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”
  - 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

- Don’t mix C-style IO and C++ stream IO!
  - Both use the same underlying OS-resources
  - Either API can leave stream in a state unexpected by the other one
- Don’t use **`printf()`** and **`scanf()`** in C++
  - At least, not in this class
  - In general, use C++ IO in C++ 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 **`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

Classes and Objects

- Objects are a tight pairing of two things:
  - State – a collection of related data values
  - Behavior – code that acts on those data values in coherent ways
  - "Objects = Data + Code"
- A class is a "blueprint" for objects
  - The class defines the state and behavior of objects of that class
  - Actually defines a new type in the language

C++ Terminology: Members

- A class is made up of members
- Data members are variable associated with the class
  - They store the class' state
  - Also called "member variables" or "fields"
- Member functions are operations the class can perform
  - The set of member functions in a class specifies its behavior
  - These functions usually involve the data members

Classes and Objects

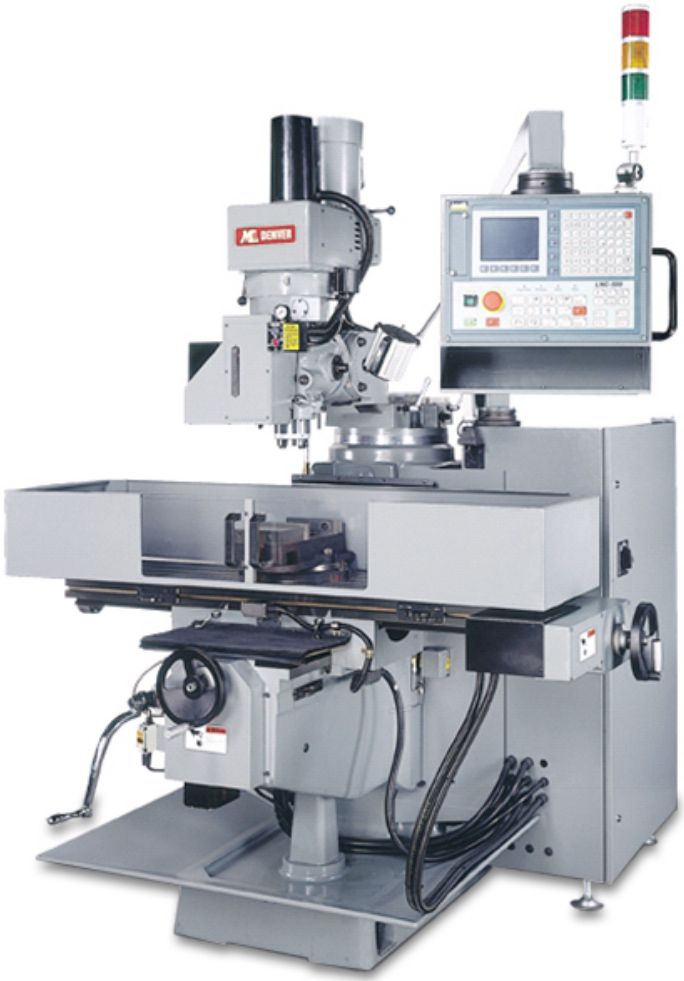
- Can have many objects of a particular class
  - Each object has its own copy of data members
  - Calling member functions on one object doesn't affect the state of other objects
- An object is an instance of a class
  - The terms "object" and "instance" are equivalent
- A class is *not* an object

Member Function Terminology

- 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 the default constructor
- Destructors clean up an instance of a class
  - This is where an instance's resources are released
  - No arguments, no return value
  - Every class has *exactly* one destructor
- 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

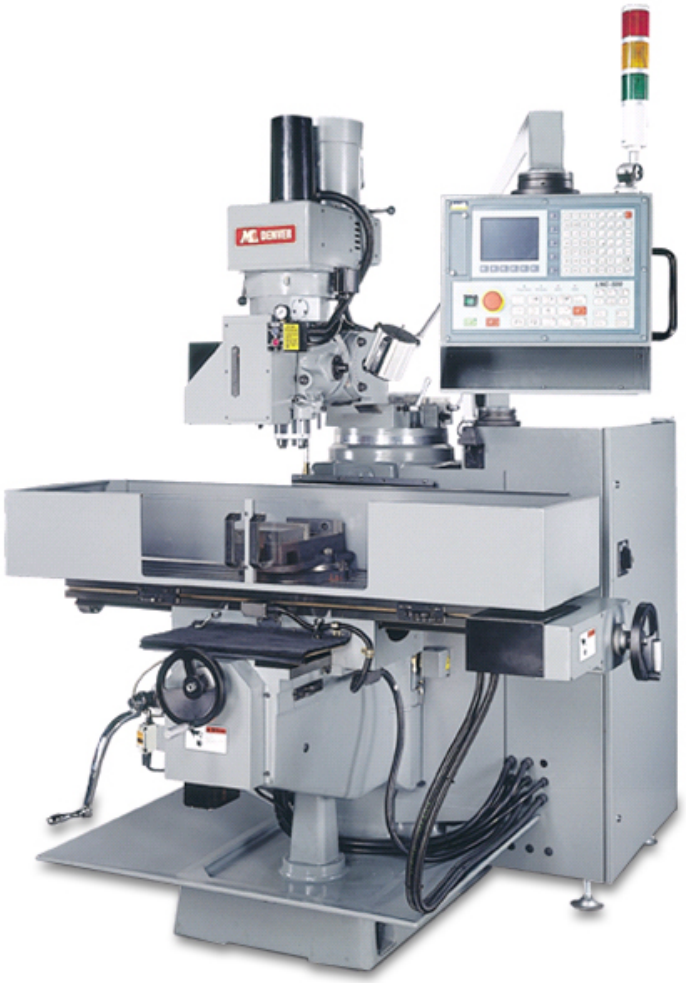
Simple Class-Design Example

- Design a class to manage a computer-controlled milling machine
- What state to maintain?
  - Current milling head coords
  - Current milling bit type
- What operations to provide?
  - Move to some location
  - Change to another milling bit



Simple Class-Design Example (2)

- State to maintain:
  - Current milling head coords
  - Current milling bit type
- Should users of class access object state directly?
  - User could change state in a way that breaks the machine!
  - The class can provide general, useful operations…
  - *The class itself* should manage the machine's state (don't leave that up to the user!)



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

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…
- Default access-level for classes is private.
- In general, other code can only access the public parts of your classes.

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`

```
#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, so doesn't do anything
}
```

Defining the Point's Behavior (continued)

```
// 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 Our Point

- 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 guts are hidden.

```
p1.x_coord = 452;    // Compiler reports an error.
```

- Don't use parentheses with default constructor!!!

```
Point p1();          // This declares a function!
```

What About The Destructor?

- In the **`Point`** class, destructor doesn't do anything!
  - **`Point`** doesn't dynamically allocate any resources
  - Compiler can clean up static resources by itself

```
// Cleans up a Point instance.
Point::~Point() {
  // no dynamic resources, so doesn't do anything
}
```

- In this case, you could even leave the destructor out
  - Compiler will generate one for you
  - Always provide a destructor if your class dynamically allocates any resources!

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 gets expensive!

C++ References

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

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

  - **p** is of type **Point &** - “reference to a Point object”
  - Using **p** is identical to using **loc** here

Characteristics of C++ References

- The referent can be changed – just like a pointer

```
// A simple, contrived example:
int i = 5;
int &j = i;    // j is a reference to i
j++;           // i == 6 now, too
```

- Much cleaner syntax than pointers!

```
// Same contrived example, with pointers:
int i = 5;
int *j = &i;  // j is a pointer to i
(*j)++;       // parentheses are necessary here
```

- Can use references to primitive variables or objects
  - **`float &f`** is a reference to a **`float`** primitive
  - **`Point &p`** is a reference to a **`Point`** object

More Characteristics of References

- Always use object references as function arguments
  - The object itself isn't copied, so it's *much* faster!
- Conversion from variable to reference is automatic

```
void outputPoint(Point &p) { ... }
...

// No extra syntax needed to pass loc to fn.
Point loc(35, -117);
outputPoint(loc);
```

- Don't use references for primitive types (usually)
  - Doesn't save any time
  - Best to avoid, except in very special circumstances

C++ References Are Constrained

- C++ references must refer to *something.*
  - Nice for functions that *require* an object
- Example: a function that takes a **`Point`** argument
  - Modify the point *in-place* to rotate it by 90°
  - Want the function to actually change the passed-in object
- Pointer way:

  ```
  void rotate90(Point *p)
  ```

  - What if **`NULL`** is passed for **`p`** ??
  - (Actually, in C++ we use **`0`** instead of **`NULL`**.)
- Reference way:

  ```
  void rotate90(Point &p)
  ```

  - Not possible to pass in nothing!

References Allow Side-Effects

- References are great when you want side-effects

```
void rotate90(Point &p) {
  double x = p.getX();
  double y = p.getY();
  p.setX(y);
  p.setY(-x);
}
...
Point f(5, 2);
rotate90(f);
```

  - `f` is changed by `rotate90()`.

- If you just want efficient function calls, beware of accidental side-effects!

Pointer and Reference Syntax

- Pointers are indicated with * in the type

```
int *pInt;            // A pointer to an integer
double *pDbl = &d;    // A pointer to a double
```

- References are indicated with & in the type

```
int &intRef = i;      // A reference to an integer
```

- The * and & symbols are reused (ugh)

```
int *pInt = &i; // Here, & means "address-of"
int j = *pInt;  // Here, * means "dereference"
int k = i * *pInt;  // First * means "multiply"
                    // Second * means "dereference"
```

- You should avoid ugly code like this. ☺

Pointer and Reference Syntax (2)

- Does **&** or ***** appear in the type specification for a variable/argument declaration?
  - It's a reference variable, or it's a pointer variable

```
int *pInt;
void outputPoint(Point &p);
```

- Does **&** appear in an expression?
  - If it's used as a unary operator, it's the address-of operator

```
double d = 36.1;
double *pDbl = &d;
```

- Does ***** appear in an expression?
  - If it's followed by a pointer, it's a "dereference" operation
  - Otherwise it's a multiplication operation

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 is the type of `q`?
  - `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++
- 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

Next Time!

- More details about classes in C++
  - (hold on to your hats…)
- C++ dynamic memory allocation
  - Destructors will quickly become *very* useful…
- Assertions
  - Have your code tell you when there are bugs.