Lecture 16: Introduction to C++
C++ – a better C, and more

- Born in 1982 – “Classes: An Abstract Data Type Facility for the C Language”
  - A collection of C macros and library routines by Bjarne Stroustrup, Bell Labs

- Evolved during 1980s to whole new language
  - But always *backward compatible* to C
    - Means any C program is also a C++ program
    - Also means C++ has “legacy problems”
  - Effective use requires abandoning some C features
    - Most notably C’s I/O library, strings, and memory allocation
Well-styled C++ programs don’t look like C programs

- Named with .cpp extension: program.cpp
- Compiling: `g++ -o program program.cpp`
  or `make program`
- New style of commenting – `//`
- Includes new way to initialize variables
  - `int x(7), sum(0);`
  - By the way, get used to initializing instead of assigning later – reason will become clear soon
  - A preview:
    `MyClass myObject(constuctor arguments);`
- A Boolean type: `bool isMax = true;`
- `main` – must be declared `int` explicitly (not the default type like C, and `main` may not be `void`)
  - And automatically returns 0 by the way
using namespace std; // huh?

- Namespaces – a way to manage global symbols

- 3 ways to access things that are in a namespace:
  - Directly – with scope resolution operator each time a name is used – `std::cout << data;`
  - Or with a using declaration – `using std::cout;`
  - Or access all names in a namespace with a using directive – `using namespace std;`

- `std` – the namespace for most C++ library tools

- Can create your own namespaces:
  - `namespace mynames { /* C++ code */ }`
```cpp
#include <iostream>

• Instead of `<stdio.h>` (actually that is `cstdio` now)
  – i.e., instead of `printf`, `scanf`, `putchar`, `FILE *`, ...
  – No worries: iostream easy to use (except for formatting)

• `cout` – technically an `ostream` object
  – Use insertion operator (`<<`) to insert data into stream
    `cout << value;` // prints value to standard output

• `cin` – technically an `istream` object
  – An extraction operator (`>>`) extracts data from stream
    `cin >> value;` // gets value from standard input`
Example: Distance Between Two Points

\[ d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]

Code at  http://www.cs.ucsb.edu/~cs16/demos/dist.cpp
// Read in two points, output distance
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

int main() {
    double x1, y1, x2, y2, s1, s2, dist;

    cout << endl;
    cout << "Coordinates of first point : ";
    cin >> x1 >> y1;
    cout << "Coordinates of second point : ";
    cin >> x2 >> y2;

    s1 = x2 - x1;
    s2 = y2 - y1;
    dist = sqrt(s1*s1 + s2*s2);

    cout << fixed << setprecision(2);
    cout << endl << "Distance between points: ";
    cout << dist << endl << endl;

    return 0;
}
Default function arguments

- Can specify parameter values in the function declaration
  
  ```c
  void func(int x = 12);
  ```

- Then function user can choose to accept the default value(s) or specify new one(s)
  
  ```c
  func(); // accept default value for x
  func(97); // specify new value for x
  ```

- Mandatory arguments are ones without default values – and these *must* come first
  
  ```c
  void func(int x = 12, int y); // illegal
  void func(int y, int x = 12); // okay
  ```

- Note also must *specify in declaration*, not definition – so compiler is sure to know about default values

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

// Returns the volume of a box.
int volume(int, int = 1, int = 1);

int main() {
    cout << "Same function called three ways:\n\n"
    << "   volume(4, 6, 2) = " << volume(4, 6, 2) << endl
    << "   volume(4, 6)    = " << volume(4, 6) << endl
    << "   volume(4)       = " << volume(4) << "\n\n";

    return 0;
}

int volume(int length, int width, int height) {
    return length * width * height;
}
```
Declaring variables, tags

• Can declare anywhere in block now
  – Even in for loop header: for (int i = 0; …)

• Tag name is the type name – no two-word types to declare struct, enum or class objects
  struct Foo { … };
    // still need it to define type
  Foo myFoo; // not struct Foo myFoo;
Dynamic memory with C++

• Use new and delete instead of malloc and free

No need to specify the size – compiler deduces it

In C++:
```c++
int *ip = new int;
// compiler knows sizeof(int)
```

But in C:
```c
int *ip = (int*)malloc(sizeof(int));
```

Best to initialize when created:
```c++
int *ip = new int(7);
```
```c
... delete ip; // free the space when done
```
C++ Classes and Objects
C++ classes

- A class is similar to a struct
  - Contains data, or member variables

- Unlike struct can also contain functions, or methods

- A class that represents a rectangle:

```cpp
class Rect{
  int length;
  int width;

public:
  Rect(int l, int w) {length = l; width = w;}
  int area() { return length*width; }
};
```
C++ accessibility

• Three levels of access:
  – Public: everyone
  – Protected: only derived (child) classes/structs
  – Private: only the containing class

• Every data and function member has an access level
C++ structures: struct vs class

• Both can have *data and functions*

• Only two differences – both in default accessibility
  – Data and functions default to public in a *struct* and to private in a *class*
  – Other difference is related to default accessibility when inheriting data or functions from a parent

• In the previous example the integers *length* and *width* defaulted to private
Declaring and defining classes

- class Foo; // just a declaration
  - Sometimes all you need – usually in header files

- class Foo { ... }; // a definition
  - Note: some or all implementation likely elsewhere

- Usual definition style is most to least accessible
  public:
  // the public interface is listed first
  int getValue();

  private:
  // most data should be here – listed last
  int value;
Implementing classes

• Usually in a separate file – foo.cpp, not foo.h
  – So #include “foo.h”

• Identify class with scope resolution operator
  
  int Foo::getValue() { return value; } 

• Implementation can include other stuff too
  – Use helper functions, data, constants, even classes
  – No worries about name conflicts in other files

• Usually one implementation file per public class
Using class instances – objects

• Declare to create on stack or global space
  – Foo foo1, foo2; // created two Foos

• Or use new to dynamically allocate
  – Foo *fooPtr = new Foo; // one more Foo

• Contact object directly with . operator
  – foo1.getValue();

• Or through a pointer with -> operator
  – fooPtr->getValue();
Constructors

• Has same name as its class, and no return type
  – Can have default values for any or all parameters

• A constructor is invoked *every* time an instance is created (whenever a class is instantiated)
  – Includes objects on the stack or dynamically allocated
  – But not invoked by creation of pointer or reference

• Compiler supplies default constructor if no constructor
  – Compiler-supplied version takes no arguments
Destructors

• A destructor is invoked whenever an object goes out of scope, or by delete for dynamically allocated objects
  – Compiler supplies one if you don’t
  – Need to write your own version if need to release dynamically allocated space or other resources

• Defined like a constructor, but with a ~ in front, and it may not take any arguments
  ~Foo();
  // syntax in header file
  Foo::~Foo() { … }
  // syntax in implementation file
Getters and Setters

• Common method of handling private members
  – Function to get value and function to set value
• Applied to the rectangle example:

```cpp
class Rect{
  int length;
  int width;

public:
  int getLength() { return length; }
  int getWidth() { return width; }
  void setLength(int l) { length = l; }
  void setWidth(int w) { width = w; }
  int area() { return length*width; }
};
```