C++ Review Session

Reference &

- C++ introduces something called a reference, which is denoted by &
  - In reality, it is more or less a fancy, easier-to-use pointer that automatically dereferences

```
void foo() {
        int x = 0;
        cout << “x: “ << x << endl;
        bar(x);
        cout << “x: “ << x << endl;
}

void bar(int &x) {
        x += 5;
}
```

Output:

```
0
5
```

Reference &

- What is actually happening here?

```
void foo() {
        int x = 0;
        cout << “x: “ << x << endl;
        bar(x);
        cout << “x: “ << x << endl;
}

void bar(int &x) {
        x += 5;
}
```

- bar is declaring that x is actually a reference
  - This means that any changes made to x in bar are also changed in methods that call bar
- bar treats x like any normal integer
- This does not require foo to pass in anything special.

Reference &

- Why should we use it?

```
void foo() {
        int x = 0;
        cout << “x: “ << x << endl;
        bar(x);
        cout << “x: “ << x << endl;
}

void bar(int &x) {
        x += 5;
}
```

- This saves you the work of having to manage and dereference pointers.
- Not only is this easier to read, there are far fewer ways to mess this up
  - You can’t have an uninitialized reference
    - No segfaults!
- This can be much more efficient for passing large objects
  - In reality, all that’s being passed is the address of x, but it acts as if you have the entire object there

Reference &

- When should we not use it?

```
void foo() {
        int x = 0;
        cout << “x: “ << x << endl;
        bar(x);
        cout << “x: “ << x << endl;
}

void bar(int &x) {
        x += 5;
}
```

- If you want the changes to x to remain in bar
- For example, if you want 0 to be printed twice, you should pass the value normally, not by reference

C++ Standard Library (stl)

C++ comes prepackaged with a lot of helpful classes and functions

We won’t go through everything, as we expect that you’ve seen templated classes like vector or map before. If you haven’t, do a quick google search for examples.

If you’re unsure, always do a quick search to see if c++ already has what you need. Nothing is worse than spending a day implementing an algorithm just to find out that c++ has a better, faster, more efficient version already.

There are three categories in the stl - Containers, Algorithms, and Iterators

Go through the other c++ slide deck, it goes into more detail than we will

STL - Templates

- Just about everything in the STL relies on templates
  - Code that doesn't rely on any particular type, but is much friendlier than void * (never again!)

```
template <typename T>
T Max (T const& a, T const& b)  {
   return a < b ? b:a;
}
```

```
int x = 5;
int y = 10;
int z = Max(x, y);
```

Templating allows max to be called on any data type that can be compared using <

This same structure can be used with classes, which you've hopefully seen:

```
vector<int> vals;
map<int, string> foo_map;
etc.
```

STL - Iterators

- Before we get any further, it’s time to tackle iterators
  - Iterators are fancy pointers that are specific to a certain class
  - Iterators are another way c++ saves you from dealing with pointers directly
- Everything in c++ STL relies on iterators
  - The containers (vector, etc.), the algorithms(find, etc.)

STL - Iterators

```
vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

for(myIntVectorIterator = myIntVector.begin();
      myIntVectorIterator != myIntVector.end();
      myIntVectorIterator++)
{
   cout<<*myIntVectorIterator<<" ";
   //Should output 1 4 8
}
```

Taken from
http://www.cprogramming.com/tutorial/stl/iterators.html

Important things to note:

- This iterator is a member of class vector
- It operates just like a pointer in some sense
  - Iterator++ moves it to the next element
- Instead of a null test, we have methods
  - begin() - returns an iterator to the start
  - end() - returns an iterator to the end
- We still dereference it with a *

Side note:

- Different iterators have different restrictions
  - This is rarely an issue

STL - “for each” (technically - Range-based for loop)

```
vector<int> vec = {0, 1, 2, 3, 4, 5};

for (int i : vec) {
        cout << i << endl;
}

for (int i : {0, 1, 2, 3, 4, 5}) {
        cout << i << endl;
}

for (int &i : vec) {
        cout << i << endl;
}
```

This is a simple and more readable way of looping through containers.

However, it can only be used over classes that have begin() and end() defined (classes with iterators), or with init-lists.

Notice that you can loop through references to the elements or through copies of the elements.

“Auto” keyword

```
vector<int> vec = {0, 1, 2, 3, 4, 5};

for (auto i : vec) {
        cout << i << endl;
}
```

```
vector<map<int, set<char> > > sillyVec;

for (auto i : sillyVec) {
        cout << “something” << endl;
}
```

Auto is a wonderfully convenient part of c++11

The compiler is able to determine from context what type should go there, saving you the typing and keeping code readable.

Classes

Classes are an expanded concept of data structures: they can contain data members, but they can also contain functions as members.

Can have a constructor function that is declared just like a regular member function

Note the difference between the class name `Rectangle` and the object name `rect`.

```
1 class Rectangle {
2     int width, height;
3   public:
4     void set_values (int,int);
5     int area (void);
6 } rect;
```

Visibility & Inheritance

public: accessible to anywhere the object is visible

private: not visible to any; accessible only from within other members of the same class (or from "friends")

protected: accessible by subclasses (and their subclasses)

friend: grants member-level access to all members in a separate class (that are not members of a class). A class cannot declare itself a friend of another class.

Scope

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition.

```
1  // classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect;
19   rect.set_values (3,4);
20   cout << "area: " << rect.area();
21   return 0;
22 }
```

>> getting an “out of scope” error?
Make sure to define in scope!

Other paraphernalia

`this` pointer: the keyword that allows an object to access its own address.

Copy constructor: Initializes one object from another of the same type; copies an object

>> getting a “copy constructor deleted” error?
Some classes don’t allow initializing a new object with an old one by “copying” it, e.g. you can’t do

```
Object a  = Object(...);
Object b = a;
```

Initialization lists

Option 1:

```
Bicycle::Bicycle(int tire_size) {
    Tire front = new Tire(tire_size);
    Tire back = new Tire(tire_size);
}
```

Option 2:

```
Bicycle::Bicycle(int tire_size) : front(tire_size), back(tire_size) {}
```

Avoids calls to default constructor.

Initialization lists (cont.)

When do we use them?
- Initializing const data members, reference data members, member objects without a default constructor (e.g. semaphore), when the parameter and data member of the same name,

Why do we use them?
- Also, better performance

>> getting a warning about order of initialization?
The names of the objects being initialized should appear in the order they are declared in the class (and after any parent class constructor call)