Useful C++ Paraphernalia

Thanks to CA Audrey Ho for writing this handout!

What is `&`? A note about reference

The *reference*, denoted by &, 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;
}
```

`bar` is declaring that `x` is actually a reference, which means that any changes made to x in `bar` are also changed in methods that call `bar`. Note that `bar` treats x like any normal integer.

Why should we use it?

- By using reference, you no longer have to deal with the pain of managing and dereferencing pointers. Not only is this easier to read, there are far fewer ways to mess this up; for instance, it's not possible to have an uninitialized reference, so goodbye segfaults!
- This can be also much more efficient for passing large objects, since in reality, all that's being passed is the address of `x`, yet it acts as if you have the entire object there.

The caveat is that, like when using a pointer, we don't want to use a reference if we want changes to the object being passed in (`x`) to stay in the helper function (`bar`). Work out for yourself how the output for the previous program would change if the parameter is no longer passed by reference.

STL: C++ Standard Library

C++ comes prepackaged with a lot of helpful classes and functions (e.g. `vector` or `map`, which you've likely seen before), so bottom line is, always do a quick search to see if C++ already has what you need before writing your own function. 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. The following is a brief overview of each; see Jerry's primer for a more in-depth take.

STL Templates

Templates are fairly ubiquitous in C++. Essentially, this is 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);
```

In this example, templating allows max to be called on any data type that can be compared using the `<` operator.

This same structure can be used with classes, which you've hopefully seen via `vector<int> vals`, `map<int, string> foo_map`, etc.

You probably won't need to write templated classes yourself for CS 110, but you will use plenty of them from the STL.

STL Iterators

Iterators are fancy pointers that are specific to a certain class. They're another way C++ saves you from dealing with pointers directly. Practically everything in C++ STL relies on iterators. Examples include `vector` (a container) and `find` (an algorithm).

```
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
}
```

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

Important things to note from this example: * This iterator is a member of class `vector` * It operates just like a pointer in some sense, i.e. `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 `*`

As an aside, note that different iterators have different restrictions, but this is rarely an issue.

Useful key terms

`for each`

This is a range-based `for` loop, which allows for a simple and more readable way of looping through containers. However, it can only be used over classes that have `begin()` and `end()` defined (i.e. classes that have iterators), or with init-lists.

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

`auto`

`auto` is a wonderfully convenient part of C++11. Basically, the compiler is able to determine from context what type should go there, which saves you the typing and keeps code readable.

Classes

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

```
class Rectangle {
    int width, height;
  public:
    void set_values(int,int);
    int area (void);
} rect;
```

Visibility and 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.

```
class Rectangle {
    int width, height;
  public:
    void set_values(int,int);
    int area(void);
} ;

void Rectangle::set_values(int x, int y) {
    width = x;
    height = y;
}

int main() {
    Rectangle rect;
    rect.set_values(3,4);
    cout << "area: " << rect.area() << endl;
```

```
    return 0;
}
```

>> Getting an "out of scope" error? Make sure to define in scope with the operator!

Other useful tidbits and common questions

What is `this`?

~~A set-up for a terrible joke~~ A pointer keyword that allows an object to access its own address

I'm getting a "copy constructor deleted" error. What's going on?

A copy constructor initializes one object from another of the same type, i.e. copies an object. However, some classes don't allow initializing a new object with an old one by "copying" it. In other words, you're being told you can't do `Object a = Object(...);` `Object b = a;`

What's an initialization list and why should I know it?

Consider the following two examples.

Example 1:

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

Example 2:

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

In a nutshell, an intialization list avoids calls to default constructor when its undesirable to do so. We use them when 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, etc. It also tends to give us better performance.

Getting a warning about order of initialization?

The names of the objects being initialized need to appear in the order they are declared in the class (and after any parent class constructor call).