Introduction to C++
Basics, Examples
Constructors & Destructors

(Most of these notes come from:  http://www.cplusplus.com/doc/tutorial/classes/ )

1 Extensions to C

C is purely procedural, with no objects, classes, or inheritance. C++ is a hybrid of C with Object Oriented characteristics.

- much stronger type checking; missing casts that produce warnings in C produce errors in C++
- the introduction of true 0-O classes
- a formal inheritance mechanism in which derived classes can specialize parent classes
- formal support for polymorphic behavior in which a derived class may override a base class method simply by providing a method of the same name
- support for function overloading in which there may be different implementations with a single function name
- an operator overloading mechanism that is analogous to function overloading
- the ability to pass parameters by reference in addition to the standard pass by value
- yet another input/output library that may be used in addition to standard and low level I/O

The class is a generalization of the C structure and can contain:

- data as well as function (method) prototypes or full implementations
- accessibility controls (private, protected, public, friend)
  - private members of a class are accessible only from within other members of the same class (or from their friends); in general, data will be private
  - protected members are accessible from members of their same class (and from their friends), but also from members of their derived classes (object will declare material to be protected so the plane can see its material)
  - public members are accessible from anywhere where the object is visible; in general, our functions will be public
  - friend functions - rights can be granted to some outside functions to access private data in a class; not preferred by O-O purists, who claim that it is a slacker’s way to compensate for bad design (which is often true!!)

2 Simple Example

```cpp
class CRectangle {
   int x, y;
   public:
      void set_values (int, int);
      int area (void);
   } rect;
```

The above defines a class (i.e. a type) called CRectangle and an object (i.e. a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level), and two member functions with public access: set_values( ) and area( ), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

```cpp
int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object’s name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```cpp
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.
3 Complete Example

// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
    public:
        void set_values (int, int);
        int area () { return (x*y); }
};

void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle recta, rectb;
    recta.set_values (3,4);
    rectb.set_values (5,6);
    cout << "recta area: " << recta.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}

recta area: 12
rectb area: 30

The most important new thing in this code is the operator of scope (:: two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside definition, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

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. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access [remember that if nothing else is said, all members of a class defined with keyword class have private access]. By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see any utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

Another thing to notice in the above program are the two instances or objects: recta and rectb. Each one has its own member variables and member functions.

Notice that the call to recta.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of object-oriented programming: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to recta.area or rectb.area. Those member functions directly used the data members of their respective objects recta and rectb.
4 Constructors and Destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area( ) before having called function set_values( )? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type - not even void.

The following is the CRectangle which includes a constructor:

```cpp
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
   int width, height;
public:
   CRectangle (int,int);
   ~CRectangle ();
   int area () { return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
   width = a;
   height = b;
}

CRectangle::~CRectangle () {
   delete width;
   delete height;
}

int main () {
   CRectangle recta (3,4);
   CRectangle rectb (5,6);
   cout << "recta area: " << recta.area() << endl;
   cout << "rectb area: " << rectb.area() << endl;
   return 0;
}
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values( ), and have included instead a constructor that performs a similar action: it initializes the values of width and height with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```cpp
CRectangle recta (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value - not even void.

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.
5 Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```cpp
class CRectangle {
    int width, height;
public:
    CRectangle ();
    CRectangle (int, int);
    int area (void) { return (width*height); }
};
CRectangle::CRectangle () {
    width = 5;
    height = 5;
}
CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}
int main () {
    CRectangle recta (3,4);
    CRectangle rectb; // notice no empty ( ) are used
    cout << "recta area: " << recta.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

6 Default Constructors

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```cpp
class CExample {
    public:
        int a,b,c;
        void multiply (int n, int m) { a=n; b=m; c=a*b; }
};
```

the compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```cpp
CExample ex;
```
But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```cpp
class CExample {
public:
    int a, b, c;
    CExample (int n, int m) { a=n; b=m; }
    void multiply () { c=a*b; }
};
```

Here we have declared a constructor that takes two parameters of type `int`. Therefore the following object declaration would be correct:

```cpp
CExample ex (2, 3);
```

But,

```cpp
CExample ex;
```

would **not** be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.