

Lab 5:

Object Orientation in C++ II

Class Constructors

In the previous program, we declared our `Box` objects, `Box1` and `Box2`, and then laboriously worked through each of the data members for each object, in order to assign an initial value to it. This is unsatisfactory from several points of view. First of all, it would be easy to overlook initializing a data member, particularly in the case of a class which had many more data members than our `Box` class. Initializing the data members of several objects of a complex class could involve pages of assignment statements. The final constraint on this approach arises when we get to defining data members of a class that don't have the attribute `public` - we won't be able to access them from outside the class anyway. There has to be a better way, and

What is a Constructor?

A class constructor is a special function in a class that is called when a new object of the class is declared. It therefore provides the opportunity to initialize objects as they are created and to ensure that data members only contain valid values.

You have no leeway in naming a class constructor - it always has the same name as the class in which it is defined. The function `Box()`, for example, is a constructor for our `Box` class. It also has no return type. It's wrong to specify a return type for a constructor; you must not even write it as `void`. The primary function of a class constructor is to assign initial values to the data elements of the class, and no return type is necessary or, indeed, permitted.

Try It Out - Adding a Constructor to the Box class

Let's extend our `Box` class to incorporate a constructor.

Example 7.1:

```
// Using a constructor
#include <iostream>
using namespace std;

class Box                // Class definition at global scope
{
    public:
        double length;    // Length of a box in inches
        double breadth;   // Breadth of a box in inches
        double height;    // Height of a box in inches

        // Constructor definition
        Box(double lv, double bv, double hv)
        {
            cout << endl << "Constructor called.";
            length = lv;           // Set values of
            breadth = bv;          // data members
            height = hv;
        }
}
```

```

        // Function to calculate the volume of a box
        double Volume()
        {
            return length * breadth * height;
        }
};

int main(void)
{
    Box Box1(78.0,24.0,18.0);    // Declare and initialize Box1
    Box CigarBox(8.0,5.0,1.0); // Declare and initialize CigarBox
    double volume = 0.0;        // Store the volume of a box here
    volume = Box1.Volume();     // Calculate volume of Box1

    cout << endl
         << "Volume of Box1 = " << volume;
    cout << endl
         << "Volume of CigarBox = "
         << CigarBox.Volume();
    cout << endl;

    return 0;
}

```

How It Works

The constructor, `Box()`, has been written with three parameters of type `double`, corresponding to the initial values for the `length`, `breadth` and `height` members of a `Box` object. The first statement in the constructor outputs a message so that we can tell when it's been called. You wouldn't do this in production programs, but, since it's very helpful in showing when a constructor is called, it's often used when testing a program. We'll use it regularly for the purposes of illustration. The code in the body of the constructor is very simple. It just assigns the arguments passed to the corresponding data members. If necessary, we could also include checks that valid, non-negative arguments are supplied and, in a real context, you probably would want to do this, but our primary interest here is in seeing how the mechanism works.

Within `main()`, we declare the object `Box1` with initializing values for the data members `length`, `breadth`, and `height`, in sequence. These are in parentheses following the object name. This uses the functional notation for initialization, which can also be applied to initializing ordinary variables of basic types. We also declare a second object of type `Box`, called `CigarBox`, which also has initializing values.

The volume of `Box1` is calculated using the member function `Volume()` as in the previous example and is then displayed on the screen. We also display the value of the volume of `CigarBox`. The output from the example is:



The first two lines are output from the two calls of the constructor, `Box()`, once for each object declared. The constructor that we've supplied in the class definition is automatically called when a `Box` object is declared, so both `Box` objects are initialized with the initializing values appearing in the declaration. These are passed to the constructor as arguments, in the sequence that they are written in the declaration. As you can see, the volume of `Box1` is the same as before and

CigarBox has a volume looking suspiciously like the product of its dimensions, which is quite a relief.

The Default Constructor

Try modifying the last example by adding the declaration for Box2 that we had previously:

```
Box Box2;           // Declare Box2 of type Box
```

Here, we've left Box2 without initializing values. When you rebuild this version of the program, you'll get the error message:

error C2512: 'Box': no appropriate default constructor available

This means that the compiler is looking for a default constructor for Box2, either one that needs no arguments, because none are specified in the constructor definition, or one whose arguments are all optional, because we haven't supplied any initializing values for the data members. Well, this statement was perfectly satisfactory in Ex6_02.cpp, so why doesn't it work now?

The answer is that the previous example used a default constructor that was supplied by the compiler, because we didn't supply one. Since in this example we *did* supply a constructor, the compiler assumed that we were taking care of everything and didn't supply the default. So, if you still want to use declarations for Box objects which aren't initialized, you have to include the default constructor yourself. What exactly does the default constructor look like? In the simplest case, it's just a constructor that accepts no arguments, it doesn't even need to do anything:

```
Box()           // Default constructor
{}             // Totally devoid of statements
```

Try It Out - Supplying a Default Constructor

Let's add our version of the default constructor to the last example, along with the declaration for Box2, plus the original assignments for the data members of Box2. We must enlarge the default constructor just enough to show that it is called. Here is the next version of the program:

```
// Supplying and using a default constructor
#include <iostream>
using namespace std;

class Box           // Class definition at global scope
{
public:
    double length;   // Length of a box in inches
    double breadth;  // Breadth of a box in inches
    double height;   // Height of a box in inches

    // Constructor definition
    Box(double lv, double bv, double hv)
    {
        cout << endl << "Constructor called.";
        length = lv;           // Set values of
        breadth = bv;          // data members
        height = hv;
    }

    // Default constructor definition
    Box()
    { cout << endl << "Default constructor called."; }

    // Function to calculate the volume of a box
    double Volume()
    {
```

```

        return length * breadth * height;
    }
};

int main(void)
{
    Box Box1(78.0,24.0,18.0); // Declare and initialize Box1
    Box Box2;                // Declare Box2 - no initial values
    Box CigarBox(8.0,5.0,1.0); // Declare and initialize CigarBox
    double volume = 0.0;       // Store the volume of a box here
    volume = Box1.Volume();    // Calculate volume of Box1

    cout << endl
         << "Volume of Box1 = " << volume;

    Box2.height = Box1.height - 10;    // Define Box2
    Box2.length = Box1.length/2.0;    // members in
    Box2.breadth = 0.25*Box1.length;  // terms of Box1

    cout << endl
         << "Volume of Box2 = "
         << Box2.Volume();


    cout << endl
         << "Volume of CigarBox = "
         << CigarBox.Volume();
    cout << endl;

    return 0;
}

```

How It Works

Now that we have included our own version of the default constructor, there are no error messages from the compiler and everything works. The program produces this output:



```

MS-DOS Batch File
Ex6_04
Constructor called.
Default constructor called.
Constructor called.
Volume of Box1 = 33696
Volume of Box2 = 6084
Volume of CigarBox = 40
Press any key to continue

```

All that our default constructor does is to display a message. Evidently, it was called when we declared the object `Box2`. We also get the correct value for the volumes of all three `Box` objects, so the rest of the program is working as it should.

One aspect of this example that you may have noticed is that we now know we can overload constructors just as we overloaded functions. We've just run an example with two constructors that differ only in their parameter list. One has three parameters of type `double` and the other has no parameters at all.

Assigning Default Values in a Constructor

When we discussed functions in C++, we saw how we could specify default values for the parameters to a function in the function prototype. We can also do this for class member functions, including constructors. If we put the definition of the member function inside the class definition, we can put the default values for the parameters in the function header. If we only

include the prototype of a function in the class definition, the default parameter value should go in the prototype.

If we decided that the default size for a `Box` object was a unit box with all sides of length 1, we could alter the class definition in the last example to this:

```
class Box                                // Class definition at global scope
{
    public:
        double length;    // Length of a box in inches
        double breadth;   // Breadth of a box in inches
        double height;    // Height of a box in inches

        // Constructor definition
        Box(double lv = 1.0, double bv = 1.0, double hv = 1.0)
        {
            cout << endl << "Constructor called.";
            length = lv;           // Set values of
            breadth = bv;          // data members
            height = hv;
        }

        // Default constructor definition
        Box()
        { cout << endl << "Default constructor called."; }

        // Function to calculate the volume of a box
        double Volume()
        {
            return length * breadth * height;
        }
};
```

If we make this change to the last example, what happens? We get another error message from the compiler, of course. We get these useful comments:

warning C4520: 'Box': multiple default constructors specified
error C2668: 'Box::Box': ambiguous call to overloaded function

This means that the compiler can't work out which of the two constructors to call - the one for which we have set default values for the parameters or the constructor that doesn't accept any parameters. This is because the declaration of `Box2` requires a constructor without parameters, and either constructor can now be called in this case. The immediately obvious solution to this is to get rid of the constructor that accepts no parameters. This is actually beneficial. Without this constructor, any `Box` object that is declared without being explicitly initialized will automatically have its members initialized to 1.

Try It Out - Supplying Default Values for Constructor Arguments

We can demonstrate this with the following simplified example:

Example 7.2:

```
// Supplying default values for constructor arguments
#include <iostream>
using namespace std;

class Box                                // Class definition at global scope
{
    public:
        double length;    // Length of a box in inches
        double breadth;   // Breadth of a box in inches
```

```

double height;    // Height of a box in inches

// Constructor definition
Box(double lv=1.0, double bv=1.0, double hv=1.0)
{
    cout << endl << "Constructor called.";
    length = lv;           // Set values of
    breadth = bv;          // data members
    height = hv;
}

// Function to calculate the volume of a box
double Volume()
{
    return length * breadth * height;
}

};

int main(void)
{
    Box Box2;              // Declare Box2 - no initial values
    cout << endl
         << "Volume of Box2 = "
         << Box2.Volume();
    cout << endl;
    return 0;
}

```

How It Works

We only declare a single uninitialized `Box` variable, `Box2`, because that's all we need for demonstration purposes. This version of the program produces the following output:



This shows that the constructor with default parameter values is doing its job of setting the values of objects that have no initializing values specified.

You shouldn't assume from the above example that this is the only, or even the recommended, way of implementing the default constructor. There will be many occasions where you won't want to assign default values in this way, in which case you'll need to write a separate default constructor. There will even be times when you don't want to have a default constructor operating at all, even though you have defined another constructor. This would ensure that all declared objects of a class must have initializing values explicitly specified in their declaration.

Using an Initialization List in a Constructor

Previously, we initialized the members of an object in the class constructor using explicit assignment. We could also have used a different technique, using what is called an **initialization list**. We can demonstrate this with an alternative version of the constructor for the class `Box`:

```

// Constructor definition using an initialization list
Box(double lv=1.0, double bv=1.0, double hv=1.0): length(lv),
                                                    breadth(bv),
                                                    height(hv)
{

```

```

        cout << endl << "Constructor called.";
    }

```

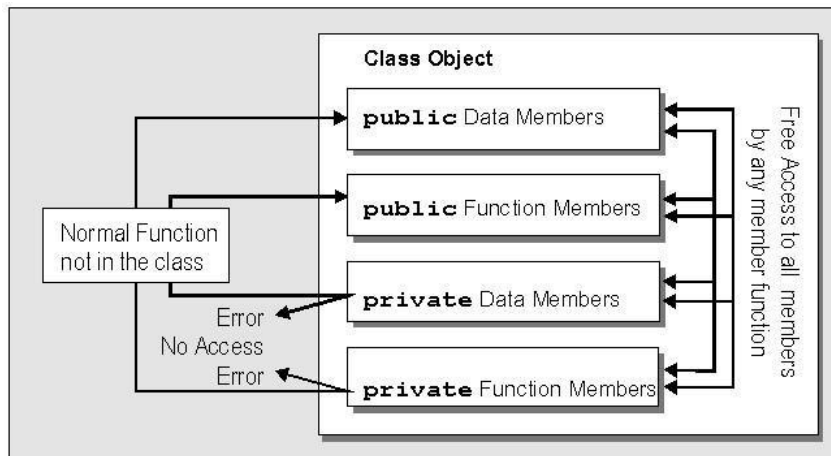
Now the values of the data members are not set in assignment statements in the body of the constructor. As in a declaration, they are specified as initializing values using functional notation and appear in the initializing list as part of the function header. The member `length` is initialized by the value of `lv`, for example. This can be rather more efficient than using assignments as we did in the previous version. If you substitute this version of the constructor in the previous example, you will see that it works just as well.

Note that the initializing list for the constructor is separated from the parameter list by a colon and that each of the initializers is separated by a comma. This technique for initializing parameters in a constructor is important, because, as we shall see later, it's the only way of setting values for certain types of data members of an object.

Private Members of a Class

Having a constructor that sets the values of the data members of a class object, but still admits the possibility of any part of a program being able to mess with what are essentially the guts of an object, is almost a contradiction in terms. To draw an analogy, once you have arranged for a brilliant surgeon such as Dr. Kildare, whose skills were honed over years of training, to do things to your insides, letting the local plumber, bricklayer or the folks from Hill Street Blues have a go hardly seems appropriate. We need some protection for our class data members.

We can get it by using the keyword `private` when we define the class members. Class members which are `private` can, in general, only be accessed by member functions of a class. There's one exception, but we'll worry about that later. A normal function has no direct means of accessing the `private` members of a class. This is shown here:



Having the possibility of specifying class members as `private` also enables you to separate the interface to the class from its internal implementation. The interface to a class is composed of the `public` members and the `public` member functions in particular, since they can provide indirect access to all the members of a class, including the `private` members. By keeping the internals of a class `private`, you can later modify them to improve performance, for example, without necessitating modifications to the code that uses the class through its public interface. To keep them safe from unnecessary meddling, it's good practice to declare data and function members of a class that don't need to be exposed as `private`. Only make `public` what is essential to the use of your class.

Try It Out - Private Data Members

We can rewrite the `Box` class to make its data members `private`.

```
// A class with private members
```

```

#include <iostream>
using namespace std;
class Box          // Class definition at global scope
{
    public:
        // Constructor definition
        Box(double lv=1.0, double bv=1.0, double hv=1.0)
        {
            cout << endl << "Constructor called.";
            length = lv;           // Set values of
            breadth = bv;          // data members
            height = hv;
        }

        // Function to calculate the volume of a box
        double Volume()
        {
            return length * breadth * height;
        }

    private:
        double length;    // Length of a box in inches
        double breadth;   // Breadth of a box in inches
        double height;    // Height of a box in inches
};

int main(void)
{
    Box Match(2.2, 1.1, 0.5); // Declare Match box
    Box Box2;                 // Declare Box2 - no initial values

    cout << endl
         << "Volume of Match = "
         << Match.Volume();

    // Uncomment the following line to get an error
    // Box2.length = 4.0;

    cout << endl
         << "Volume of Box2 = "
         << Box2.Volume();
    cout << endl;

    return 0;
}

```

How It Works

The definition of the class `Box` now has two sections. The first is the `public` section containing the constructor and the member function `Volume()`. The second section is specified as `private` and contains the data members. Now the data members can only be accessed by the member functions of the class. We don't have to modify any of the member functions - they can access all the data members of the class anyway. However, if you uncomment the statement in the function `main()`, assigning a value to the member `length` of the object `Box2`, you'll get a compiler error message confirming that the data member is inaccessible.

A point to remember is that using a constructor or a member function is now the only way to get a value into a private data member of an object. You have to make sure that all the ways in which you might want to set or modify private data members of a class are provided for through member functions.

We could also put functions into the `private` section of a class. In this case, they can only be called by other member functions. If you put the function `Volume()` in the `private` section, you will get a compiler error from the statements that attempt to use it in the function `main()`. If you put the constructor in the `private` section, you won't be able to declare any members of the class.

The above example generates this output:



This demonstrates that the class is still working satisfactorily, with its data members defined as having the access attribute `private`. The major difference is that they are now completely protected from unauthorized access and modification.

If you don't specify otherwise, the default access attribute which applies to members of a class is `private`. You could, therefore, put all your private members at the beginning of the class definition and let them default to `private` by omitting the keyword. However, it's better to take the trouble to explicitly state the access attribute in every case, so there can be no doubt about what you intend.

Of course, you don't have to make all your data members `private`. If the application for your class requires it, you can have some data members defined as `private` and some as `public`. It all depends on what you're trying to do. If there's no reason to make members of a class `public`, it is better to make them `private` as it makes the class more secure. Ordinary functions won't be able to access any of the `private` members of your class.

Accessing private Class Members

On reflection, declaring all the data members of a class as `private` might seem rather extreme. It's all very well protecting them from unauthorized modification, but that's no reason to keep their values a secret. What we need is a Freedom of Information Act for `private` members.

You don't need to start writing to your state senator to get it - it's already available to you. All you need to do is to write a member function to return the value of a data member. Look at this member function for the class `Box`:

```
inline double Box::GetLength(void)
{
    return length;
}
```

Just to show how it looks, this has been written as a member function definition which is external to the class. We've specified it as `inline`, since we'll benefit from the speed increase, without increasing the size of our code too much. Assuming that you have the declaration of the function in the `public` section of the class, you can use it by writing this statement:

```
len = Box2.GetLength();    // Obtain data member length
```

All you need to do is to write a similar function for each data member that you want to make available to the outside world, and their values can be accessed without prejudicing the security of the class. Of course, if you put the definitions for these functions within the class definition, they will be `inline` by default.

The Default Copy Constructor

Suppose we declare and initialize a `Box` object `Box1` with this statement:

```
Box Box1(78.0, 24.0, 18.0);
```

We now want to create another `Box` object, identical to the first. We would like to initialize the second `Box` object with `Box1`.

Try It Out - Copying Information Between Instances

We can try this out with the `main()` function which follows:

Example 7.3:

```
// Initializing an object with an object of the same class
#include <iostream>
using namespace std;

class Box          // Class definition at global scope
{
    public:
        // Constructor definition
        Box(double lv=1.0, double bv=1.0, double hv=1.0)
        {
            cout << endl << "Constructor called.";
            length = lv;                // Set values of
            breadth = bv;                // data members
            height = hv;
        }

        // Function to calculate the volume of a box
        double Volume()
        {
            return length * breadth * height;
        }

    private:
        double length;    // Length of a box in inches
        double breadth;   // Breadth of a box in inches
        double height;    // Height of a box in inches
};

int main(void)
{
    Box Box1(78.0, 24.0, 18.0);
    Box Box2 = Box1;        // Initialize Box2 with Box1

    cout << endl
         << "Box1 volume = " << Box1.Volume()
         << endl
         << "Box2 volume = " << Box2.Volume();

    cout << endl;
    return 0;
}
```

How It Works

This example will produce this:



Clearly, the program is working as we would want, with both boxes having the same volume. However, as you can see from the output, our constructor was called only once for the creation of `Box1`. But how was `Box2` created? The mechanism is similar to the one that we experienced when we had no constructor defined and the compiler supplied a default constructor to allow an object to be created. In this case, the compiler generates a default version of what is referred to as a **copy constructor**.

A copy constructor does exactly what we're doing here - it creates an object of a class by initializing it with an existing object of the same class. The default version of the copy constructor creates the new object by copying the existing object, member by member.

This is fine for simple classes such as `Box`, but for many classes - classes that have pointers or arrays as members for example - it won't work properly. Indeed, with such classes it can create serious errors in your program. In these cases, you need to create your own class copy constructor.

Destructor:

A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Example 7.4:

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

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

    int area ()
    {return (width * height);}

    ~CRectangle () {

    cout<<"deleting object..."<<endl;
```

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

Note:

Before doing your lab exercises run the examples.

Exercises will be provided by the instructors in the lab.