

## Lab 4:

### Object Orientation in C++ I

---

#### Data Types, Objects, Classes and Instances

So far, we've learnt that C++ lets you create variables which can be any of a range of basic data types: int, long, double and so on. However, the variables of the basic types don't allow you to model real-world objects (or even imaginary objects) adequately. It's hard to model a box in terms of an int, for example; what we need is something that allows us to collect together the various attributes of an object. In C++, we can do this very easily using classes.

You could define variables, length, breadth and height to represent the dimensions of the box and bind them together as members of a Box class, as follows:

```
class Box
{
public:
    double length;
    double breadth;
    double height;
};
```

This code actually creates a *new type*, called Box. The keyword class defines Box as such, and the elements that make up this class are defined within the curly braces. Note that each line defining an element of the class is terminated by a semicolon, and that a semicolon also appears after the closing brace. The elements length, breadth and height are referred to as **data members**. At the top of the class definition, you can see we have put the keyword public - this just means that the data members are generally accessible from outside the class. You may, however, place restrictions on the accessibility of class members, and we'll see how this works a bit later in this chapter.

With this definition of a new data type called Box, you can go ahead and define variables of this type just as you did with variables of the basic types:

```
Box myBox;           //Declare a variable myBox of type Box
```

Once we've defined the class Box, the declaration of variables of this type is quite standard. The variable myBox here is also referred to as an **object** or an **instance** of the class Box.

With this definition of a new data type called Box, you can go ahead and define variables of this type just as you did with variables of the basic types. You can then create, manipulate and destroy as many Box objects as you need to in your program. This means that you can model objects using classes and write your programs around them. So - that's object-oriented programming all wrapped up then?

Well, not quite. You see, object-oriented programming (OOP) is based on a number of foundations (famously *encapsulation*, *polymorphism* and *inheritance*) and what we have seen so far doesn't quite fit the bill. Don't worry about what these terms mean for the moment - we'll be exploring these ideas in more detail as we learn more about what you can do with classes.

#### First Class

The notion of class was invented by an Englishman to keep the general population happy. It derives from the theory that people who knew their place and function in society would be much more secure and comfortable in life than those who did not. The famous Dane, Bjarne Stroustrup, who invented C++,

undoubtedly acquired a deep knowledge of class concepts while at Cambridge University in England, and appropriated the idea very successfully for use in his new language.

The idea of a class in C++ is similar to the English concept, in that each class usually has a very precise role and a permitted set of actions. However, it differs from the English idea, because class in C++ has largely socialist overtones, concentrating on the importance of working classes. Indeed, in some ways it is the reverse of the English ideal, because, as we shall see, working classes in C++ often live on the backs of classes that do nothing at all.

## Operations on Classes

In C++ you can create new data types as classes to represent whatever kinds of objects you like. As you'll come to see, classes aren't limited to just holding data; you can also define member functions that act on your objects, or even operations that act between objects of your classes using the standard C++ operators. You can define the class `Box`, for example, so that the following statements work and have the meanings you want them to have:

```
Box Box1;
Box Box2;
if(Box1 > Box2)    // Fill the larger box
    Box1.Fill();
else
    Box2.Fill();
```

You could also implement operations as part of the `Box` class for adding, subtracting or even multiplying boxes - in fact, almost any operation to which you could ascribe a sensible meaning in the context of boxes.

We're talking about incredibly powerful medicine here and it constitutes a major change in the approach that we can take to programming. Instead of breaking down a problem in terms of what are essentially computer-related data types (integer numbers, floating point numbers and so on) and then writing a program, we're going to be programming in terms of problem-related data types, in other words classes. These classes might be named `Employee`, or `Cowboy`, or `Cheese` or `Chutney`, each defined specifically for the kind of problem that you want to solve, complete with the functions and operators that are necessary to manipulate instances of your new types.

Program design now starts with deciding what new application-specific data types you need to solve the problem in hand and writing the program in terms of operations on the specifics that the problem is concerned with, be it `Coffins` or `Cowpokes`.

## Terminology

Let's summarize some of the terminology that we will be using when discussing classes in C++:

- A **class** is a user-defined data type
- **Object-oriented programming** is the programming style based on the idea of defining your own data types as classes
- Declaring an object of a class is sometimes referred to as **instantiation** because you are creating an **instance** of a class
- Instances of a class are referred to as **objects**
- The idea of an object containing the data implicit in its definition, together with the functions that operate on that data, is referred to as **encapsulation**

When we get into the detail of object-oriented programming, it may seem a little complicated in places, but getting back to the basics of what you're doing can often help to make things clearer, so always keep in mind what objects are really about. They are about writing programs in terms of the objects that are

specific to the domain of your problem. All the facilities around classes in C++ are there to make this as comprehensive and flexible as possible. Let's get down to the business of understanding classes.

## Understanding Classes

A class is a data type that you define. It can contain data elements, which can either be variables of the basic types in C++ or other user-defined types. The data elements of a class may be single data elements, arrays, pointers, arrays of pointers of almost any kind or objects of other classes, so you have a lot of flexibility in what you can include in your data type. A class can also contain functions which operate on objects of the class by accessing the data elements that they include. So, a class combines both the definition of the elementary data that makes up an object and the means of manipulating the data belonging to individual instances of the class.

The data and functions within a class are called **members** of the class. Funnily enough, the members of a class that are data items are called **data members** and the members that are functions are called **function members** or **member functions**. The member functions of a class are also sometimes referred to as **methods**, although we will not be using this term.

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. It's much the same as if you wrote a description of the basic type `double`. This wouldn't be an actual variable of type `double`, but a definition of how it's made up and how it operates. To create a variable, you need to use a declaration statement. It's exactly the same with classes, as you will see.

## Defining a Class

Let's look again at the class we started talking about at the start of the chapter - a class of boxes. We defined the `Box` data type using the keyword `class` as follows:

```
class Box
{
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
};
```

The name that we've given to our class appears following the keyword and the three data members are defined between the curly braces. The data members are defined for the class using the declaration statements that we already know and love, and the whole class definition is terminated with a semicolon. The names of all the members of a class are local to a class. You can therefore use the same names elsewhere in a program without causing any problems.

## Access Control in a Class

The keyword `public` looks a bit like a label, but in fact it is more than that. It determines the access attributes of the members of the class that follow it. Specifying the data members as `public` means that these members of an object of the class can be accessed anywhere within the scope of the class object. You can also specify the members of a class as `private` or `protected`. In fact, if you omit the access specification altogether, the members have the default attribute, `private`. We shall look into the effect of these keywords in a class definition a bit later.

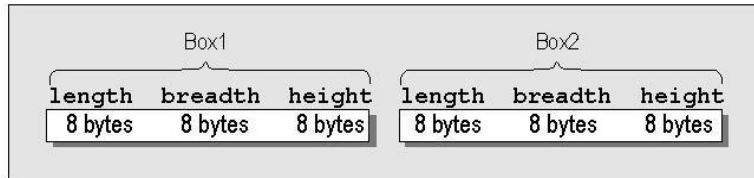
Remember that all we have defined so far is a class, which is a data type. We haven't declared any objects of the class. When we talk about accessing a class member, say `height`, we're talking about accessing the data member of a particular object, and that object needs to be declared somewhere.

## Declaring Objects of a Class

We declare objects of a class with exactly the same sort of declaration that we use to declare objects of basic types. We saw this at the beginning of the chapter. So, we could declare objects of our class, `Box`, with these statements:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects `Box1` and `Box2` will, of course, have their own data members. This is illustrated here:



The object name `Box1` embodies the whole object, including its three data members. They are not initialized to anything, however - the data members of each object will simply contain junk values, so we need to look at how we can access them for the purpose of setting them to some specific values.

### Accessing the Data Members of a Class

The data members of objects of a class can be referred to using the **direct member access operator** (`.`). So, to set the value of the data member `height` of the object `Box2` to, say, 18.0, we could write this assignment statement:

```
Box2.height = 18.0;          // Setting the value of a data member
```

We can only access the data member in this way, in a function outside the class, because the member `height` was specified as having `public` access. If it wasn't defined as `public`, this statement would not compile. We'll see more about this shortly.

### Try It Out - Your First Use of Classes

Let's have a go at creating a `Box` class and setting the values of its data members. We'll try it out in the following console application:

#### Example 6.1:

```
// Creating and using boxes
#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
};

int main(void)
{
    Box Box1;          // Declare Box1 of type Box
    Box Box2;          // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of a box here
    Box1.height = 18.0; // Define the values
    Box1.length = 78.0; // of the members of
    Box1.breadth = 24.0; // the object Box1
    Box2.height = Box1.height - 10; // Define Box2
    Box2.length = Box1.length/2.0; // members in
    Box2.breadth = 0.25*Box1.length; // terms of Box1
```

```

// Calculate volume of Box1
volume = Box1.height * Box1.length * Box1.breadth;
cout << endl
    << "Volume of Box1 = " << volume;
cout << endl
    << "Box2 has sides which total "
    << Box2.height + Box2.length + Box2.breadth
    << " inches.";
cout << endl    // Display the size of a box in memory
    << "A Box object occupies "
    << sizeof Box1 << " bytes.";
cout << endl;

return 0;
}

```

## How It Works

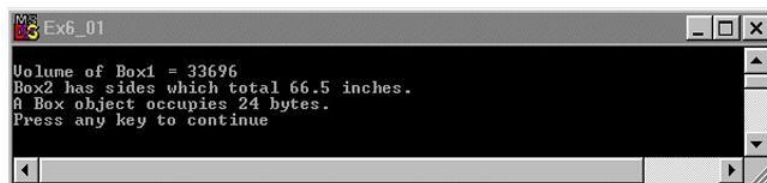
Everything here works as we would have expected from our experience with structures. The definition of the class appears outside of the function `main()` and, therefore, has global scope. This enables objects to be declared in any function in the program and causes the class to show up in the ClassView once the program has been compiled.

We've declared two objects of type `Box` within the function `main()`, `Box1` and `Box2`. Of course, as with variables of the basic types, the objects `Box1` and `Box2` are local to `main()`. Objects of a class obey the same rules with respect to scope as variables declared as one of the basic types (such as the variable `volume`, which is used in this example).

The first three assignment statements set the values of the data members of `Box1`. We define the values of the data members of `Box2` in terms of the data members of `Box1` in the next three assignment statements.

We then have a statement which calculates the volume of `Box1` as the product of its three data members. This value is then output to the screen. Next, we output the sum of the data members of `Box2` by writing the expression for the sum of the data members directly in the output statement. The final action in the program is to output the number of bytes occupied by `Box1`, which is produced by the operator `sizeof`.

If you run this program, you should get this output:



The last line shows that the object `Box1` occupies 24 bytes of memory, which is a result of it having 3 data members of 8 bytes each. The statement which produced the last line of output could equally well have been written like this:

```

cout << endl    // Display the size of a box in memory
    << "A Box object occupies "
    << sizeof (Box) << " bytes.";

```

Here, we've used the type name between parentheses, rather than a specific object name - this is standard syntax for the sizeof operator.

This example has demonstrated the mechanism for accessing the public data members of a class. It also shows that they can be used in exactly the same way as ordinary variables.

## Using Pointers to Objects

As you might expect, we can create a pointer to variable of a Box type. The declaration for this is just what you might expect:

```
Box* pBox = &aBox
```

Assuming we have already declared an object, aBox, of type Box, then this line simply declares a pointer to type Box and initializes it with the address of aBox. We could now access the members of aBox through the pointer pBox, using statements like this:

```
(*pBox).length = 10;
```

The parenthesis to de-reference the pointer here are essential, since the member access operator takes precedence over the de-reference operator. Without the parenthesis, we would be attempting to treat the pointer like an object and to de-reference the member, so the statement would not compile.

This method of accessing a member of the class via a pointer looks rather clumsy. Since this kind of operation crops up a lot in C++, the language includes a special operator to enable you to express the same thing in a much more readable and intuitive form. It is called the **indirect member access operator**, and it is specifically for accessing members of an object through a pointer. We could use it to re-write the statement to access the length member of aBox through the pointer pBox:

```
pBox->length = 10;
```

The operator looks like a little arrow and is formed from a minus sign followed by the symbol for 'greater than'. It's much more expressive of what's going on, isn't it? We'll be seeing a lot more of this operator as we go along and learn more about classes. We're now ready to break new ground by taking a look at member *functions* of a class.

## Member Functions of a Class

A member function of a class is a function that has its definition or its prototype within the class definition. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

## Try It Out - Adding a Member Function to Box

To see how accessing the members of the class works, let's create an example extending the Box class to include a member function.

### Example 6.2:

```
// Calculating the volume of a box with a member function
#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
    // Function to calculate the volume of a box
    double Volume(void)
    {
```

```

    return length * breadth * height;
}
};

int main(void)
{
    Box Box1;        // Declare Box1 of type Box
    Box Box2;        // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of a box here
    Box1.height = 18.0; // Define the values
    Box1.length = 78.0; // of the members of
    Box1.breadth = 24.0; // the object Box1
    Box2.height = Box1.height - 10; // Define Box2
    Box2.length = Box1.length/2.0; // members in
    Box2.breadth = 0.25*Box1.length; // terms of Box1

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

    cout << endl
        << "A Box object occupies "
        << sizeof Box1 << " bytes.";
    cout << endl;

    return 0;
}

```


## How It Works

The new code that we've added to the class definition is highlighted. It's just the definition of the function `Volume()`, which is a member function of the class. It also has the same access attribute as the data members: `public`. This is because every class member declared following an access attribute will have that access attribute, until another one is specified within the class definition. The function `Volume()` returns the volume of a `Box` object as a value of type `double`. The expression in the return statement is just the product of the three data members of the class.

*There's no need to qualify the names of the class members in any way when accessing them in member functions. The unqualified member names automatically refer to the members of the object that is current when the member function is executed.*

The member function `Volume()` is used in the highlighted statements in `main()`, after initializing the data members (as in the first example). Using the same name for a variable in `main()` causes no conflict or problem. You can call a member function of a particular object by writing the name of the object to be processed, followed by a period, followed by the member function name. As we noted above, the function will automatically access the data members of the object for which it was called, so the first use of `Volume()` calculates the volume of `Box1`. Using only the name of a data member will always refer to the member of the object for which the member function has been called.

The member function is used a second time directly in the output statement to produce the volume of `Box2`. If you execute this example, it will produce this output:



```
Ex6_02
Volume of Box1 = 33696
Volume of Box2 = 6084
A Box object occupies 24 bytes.
Press any key to continue
```

Note that the Box object is still the same number of bytes. Adding a function member to a class doesn't affect the size of the objects. Obviously, a member function has to be stored in memory somewhere, but there's only one copy regardless of how many class objects have been declared, and it isn't counted when the operator `sizeof` produces the number of bytes that an object occupies.

The names of the class data members in the member function automatically refer to the data members of the specific object used to call the function, and the function can only be called for a particular object of the class. In this case, this is done by using the direct member access operator (`.`) with the name of an object.

*If you try to call a member function without specifying an object name, your program will not compile.*

## Positioning a Member Function Definition

A member function definition need not be placed inside the class definition. If you want to put it outside the class definition, you need to put the prototype for the function inside the class. If we rewrite the previous class with the function definition outside, then the class definition looks like 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
    double Volume(void); // Member function prototype
};
```

Now we need to write the function definition, but there has to be some way of telling the compiler that the function belongs to the class Box. This is done by prefixing the function name with the name of the class and separating the two with the **scope resolution operator**, `::`, which is formed from two successive colons. The function definition would now look like this:

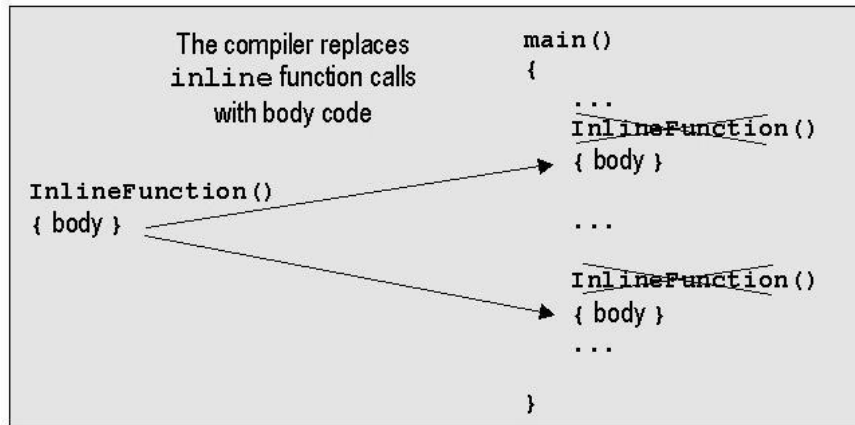
```
// Function to calculate the volume of a box
double Box::Volume(void)
{
    return length * breadth * height;
}
```

It will produce the same output as the last example. However, it isn't exactly the same program. In the second case, all calls to the function are treated in the way that we're already familiar with. However, when we defined the function within the definition of the class in `Ex6_02.cpp`, the compiler implicitly treated the function as an **inline function**.

## Inline Functions

With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. This avoids much of the overhead of calling the function and, therefore, speeds up your code. This is illustrated here:





*Of course, the compiler takes care of ensuring that expanding a function inline doesn't cause any problems with variable names or scope.*

The compiler may not always be able to insert the code for a function inline (such as with recursive functions or functions for which you have obtained an address), but generally it will work. It's best used for very short, simple functions, such as our function `Volume()`, because they will execute faster and will not significantly increase the size of the executable module.

With the function definition outside of the class definition, the compiler treats the function as a normal function and a call of the function will work in the usual way. However, it's also possible to tell the compiler that, if possible, we would like the function to be considered as inline. This is done by simply placing the keyword `inline` at the beginning of the function header. So, for our function, the definition would be as follows:

```

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

```

With this definition for the function, the program would be exactly the same as the original. This allows you to put the member function definitions outside of the class definition, if you so choose, and still retain the speed benefits of inlining.

You can apply the keyword `inline` to ordinary functions in your program that have nothing to do with classes and get the same effect. However, remember that it's best used for short, simple functions.

We now need to understand a little more about what happens when we declare an object of a class.

**Note:**

***Before doing your lab exercises run the examples.***

**Exercises will be provided by the instructors in the lab.**