**Chapter 6**

Table of Contents

[Classes 2](#_Toc49508179)

[Objects and Functions 4](#_Toc49508180)

[Destructors 8](#_Toc49508181)

[Objects as Function Arguments 9](#_Toc49508182)

[Default Copy Constructor 11](#_Toc49508183)

[Returning an Object from A Function 13](#_Toc49508184)

[Structures and Classes 15](#_Toc49508185)

[Classes, Objects and Memory 16](#_Toc49508186)

[Static Class Data 17](#_Toc49508187)

[Static Functions 19](#_Toc49508188)

[Constant Functions 20](#_Toc49508189)

[Constant Objects 21](#_Toc49508190)

## Classes

class SomeClass  
{  
private:  
 int someData;  
 int someOtherData;  
public:  
 void setData (int val)  
 {  
 someData = val;  
 }  
 int getData()  
 {  
 return someData;  
 }  
};

C++

The above code is one example of a class. It is declared independently, outside of any functions. There are many things to notice here.

Starting from the top, note that the class name SomeClass, starts with a capital letter. This is different from how functions and variables are named, and although not required, it is convention.

Inside the class, we have to access modifiers, private and public. Everything written under the private access modifier is accessible only from within the class, but everything written under the public access modifier is accessible from anywhere in the program. Notice that all the data is declared private, and only functions are declared public. The functions are created in such a way so as to allow the rest of the program to use those functions to interact with the data. This is generally the way object-oriented programming is written, with the data inside a class only being accessible from the outside through the member functions of that class.

If a variable or function is not declared private or public, it is private by default. We can also declare functions to be private instead of public. These will be shown in the next example. It is also acceptable to just declare a function within a class and define it later on, i.e. prototyping. This will also be shown in the next example.

Lastly, but perhaps most importantly, notice the semicolon at the end. Yes, classes are declared in exactly the same way as structures.

## Objects and Functions

Now look at the following code. This is an entire program that can be used as a counter.

#include<iostream>  
using namespace std;  
  
class Counter  
{  
 int counter;  
public:  
 void resetCount();  
 void incrementCount();  
 int getCount();  
};  
void Counter :: resetCount()  
{  
 counter = 0;  
}  
  
  
void Counter :: incrementCount()  
{  
 counter++;  
}

int Counter :: getCount()  
{  
 return counter;  
}  
  
int main()  
{  
 Counter c1, c2, c3;  
 *// c1.counter = 0;* c1.resetCount();  
 cout<<c1.getCount()<<endl;  
 c1.incrementCount();  
 cout<<c1.getCount()<<endl;  
}

C++

The above code implements a few of the things already mentioned, and a few new things as well.

Firstly, inside the class, we did not declare whether counter is public or private. It will be automatically considered private (this is proven later on in the code).

We have also used prototyping, declaring all three functions inside the class but defining them outside. This is done using the class name along with the operator :: as shown. :: is known as the scope resolution operator. Remember that it can also be used to declare the namespace for library functions. It has other uses as well. Prototyping should be used depending on whether it improves readability. In this case, the functions do very small tasks and their names make the tasks obvious, so it is helpful to simply declare the functions inside the class, so that the entire class can be understood at a glance. For functions that do a number of tasks which are not so obvious, it could be more helpful to define the entire function inside the class, so as to avoid confusion. Note that this method cannot be used to create new functions.

Inside the main function, we first create three objects of the Counter class, c1, c2 and c3. Now, in order to increment the variable counter, we must first give it a value. Notice that there is one line of code here that is commented out. Attempting to set the value of counter directly like this will cause an error, since it is a private variable. This proves two things, firstly that the variable counter was automatically made private, and secondly that private variables really cannot be accessed from outside their class.

We set the value of counter using the resetCounter() function. In this scenario, every time we create an object of the Counter class, we must set the value of counter before doing anything else. This raises two questions, firstly, why this was not done inside the class, and secondly, whether it can be done automatically.

To answer the first question, we must remember that a class is just a blueprint. It holds information about what type of data an object of the class will contain, but it does not contain any data in itself, i.e. it does not take up any space in memory. When we create the objects c1, c2 and c3, then memory is allocated to each of those objects separately. Thus, attempting to give counter a value inside the class will cause an error.

To answer the second question, yes, the process can be automated. This is done with the help of something called a constructor function. A constructor function has no return type, and has the same name as the class it belongs to. It is run automatically, as soon as an object of that class is created, and can be used exactly like a normal function. It may or may not be given parameters, but how parameters can be passed to a constructor function will be discussed later.

To create a constructor function, we must either declare or define it publicly inside the class. Assuming it was only declared, this is how it can be defined from outside the class:

Counter :: Counter()  
{  
 resetCount();  
 cout<<”Constructor function has run!”<<endl;  
}

C++

This means we no longer have to use the resetCount() function when creating an object of the Counter class. This gives us an opportunity to make it private inside our class (by declaring it before the public keyword). We have also added a line inside out constructor function, just to make sure it works. Make these changes to the code and run it again to see that the line prints three times before anything else, since we created three objects of the Counter class at the beginning of our main function.

The particular use we have put our constructor function to, initializing a variable, is so common, that there is a special way to do it, known as an initialization list. In order to set the variable counter to 0, we can declare the constructor function like this:

Counter :: Counter(): counter (0)  
{}

C++

This allows us to remove the resetCounter() function entirely, and also gives us an empty constructor function, which is perfectly valid. Since the way this function is defined is so very not confusing, colours have been added to help explain how it works. The green bit tells us which class to go to, the white bit tells us which function to go to (and since the function and class names are the same, we know it is a constructor function) and finally, the purple bit tells us which variable to go to and set a value (notice that a single colon is used, not the :: operator). We can also initialize multiple variables like this:

Counter :: Counter() : counter1 (0), counter2 (1){}

C++

## Destructors

When a function, even the main function, terminates, the objects created within that function are all destroyed. Destructors are functions that execute automatically just before the object related to them is destroyed. They are the opposite of constructors. Destructors cannot have a return type or be given any parameters since they are never actually called.

Class Counter  
{  
 int counter;  
  
public:  
 Counter() : counter(0)  
 {  
 cout<<”Constructor\n”;  
 }  
  
 ~Counter()  
 {  
 cout<<”Destructor\n”;  
 }  
};

C++

Interestingly, the destructor functions follow the ‘Last in First Out’ process, meaning the last object created will be destroyed first, and its destructor function will be run first.

## Objects as Function Arguments

Consider the following program:

#include<iostream>  
using namespace std;  
  
class Distance  
{  
 int feet;  
 float inches;  
  
public:  
 Distance() : feet(0), inches(0.0F)  
 {}  
 Distance(int f, float i) : feet(f), inches(i)  
 {}  
  
 void add(Distance d1, Distance d2)  
 {  
 inches = d1.inches + d2.inches;  
 feet = d1.feet + d2.feet;  
 if (inches >= 12.0F)  
 {  
 feet++;  
 inches -= 12.0F;  
 }  
 }  
  
 void display()  
 {  
 cout<<feet<<" Feet "<<inches<<" Inches"<<endl;  
 }  
};  
  
int main()  
{  
 Distance d0, d1(5, 4.2), d2(6, 1.0);  
 d0.add(d1, d2);  
 d1.display();  
 d2.display();  
 d0.display();  
}

C++

The class stores a value of distance in feet and inches. There are two constructor functions, one that sets the distance to 0 by default, and another that allows a custom value to be set when an object is created. We also have a function that displays the value.

One important thing to notice is that the constructor initialized inches, which is a float, to 0.0F. The F at the end is important since there is a small possibility that the compiler will think it is a double value otherwise. This may cause problems if there are overloaded functions with one taking a double as input and another taking a float as input. It is good practice to specify floats this way.

The add function is the important one here. An object of the Distance class has this function defined with it, that allows it to take two other objects of the same Distance class, and set the sum of their distances to its own distance. This is a reminder that when we create a class, we can use it exactly like a variable type. We are passing objects to a function in the same way in which we pass variables.

## Default Copy Constructor

Think about the Counter class we created in a previous lecture. Let’s say it has a constructor that takes an input parameter and sets the initial value of count with it. Inside the main function we have the following code:

Counter c1(5);  
Counter c2;  
c2 = c1;

C++

We created a new object of the Counter class, c1, and set the initial value of c1.count to 5 using the constructor. Next, we created a second object of the same class, c2, and declared c2 to be equal to c1, meaning all of the data of c1 is copied to c2. This makes sense, since class objects can essentially be used like variables.

Next, we create another object like this:

Counter c3(c1);

C++

This makes less sense. The constructor we have set up does not take an object as a parameter, but we are still passing one. This line actually performs the same action as the previous line, copying all the data of c1 to c3. This is done with the help of something called the Default Copy Constructor, which exists specifically for this purpose. We cannot manually call it, and it comes automatically with every class. When we declared c2 = c1, the same Copy Constructor was actually being called.

If we had two values in the class and needed to initialize both of them, both methods, using the equal operator and using the Default Copy Construct, would support this as well.

The Default Copy Constructor can be overloaded to perform custom actions as well. This is done like this:

Counter (const Counter& c)  
{  
 counter = c.counter + 5;  
};

C++

Notice that we are passing the object by reference and not by value. Passing something by value causes a copy of that thing to be created for the use of that function. If we pass something by value to a copy constructor, a copy would have to be made, but making that copy would require the constructor. This causes an endless loop. When we pass it by reference, the original object is being passed, just under a different name.

Since the actual object is being passed and not a copy, we will be able to edit it as well. Thus, we are using the const keyword. This ensures that the object being passed to the constructor does not have its values edited in any way.

We are also accessing the variable counter of the object being passed, which is a private variable. This is possible since the function is part of the class.

Editing the Default Copy Constructor will cause the changes to be reflected on statements like c2 = c1 as well.

## Returning an Object from A Function

Just like we can pass an object as an argument to a function, we can also return an object from a function. Remember the Distance class we created in a previous lecture. Say we want to create a new function for that class. Inside the class, we declare the function:

Distance add (Distance dist);

C++

This tells the compiler that we want to create a function called add with an input argument that is an object of the class Distance, and that the return type of the function will be an object of the class Distance. Notice how we are using Distance as if it were a variable type like int or char.

Next, we define the function outside of the class (it could have been defined inside as well):

Distance Distance :: add (Distance dist1)  
{  
 Distance temp;  
 temp.inches = dist1.inches + inches;  
 temp.feet = dist1.feet + feet;  
 if (temp.inches >= 12.0)  
 {  
 temp.inches -= 12.0;  
 temp.feet++;  
 }  
 return temp;  
}

C++

And inside the main function we call it like this:

d3 = d1.add(d2);

C++

The first line of the function definition can seem a little confusing. The first keyword Distance refers to the return type of the function. The second keyword Distance tells the compiler to look inside the Distance class, and then find the add function. The third keyword Distance, inside the braces, is for the input parameter type.

What this does is, it goes to the add function inside the d1 object, using d2 as its parameter. It then adds the distance of d2 to d1 (since d1 is the object through which we called the function, inches and feet refer to the distance of d1), and store it in a temporary variable of type Distance. Finally, the temporary variable is returned to the main function, causing it to be set to d3.

Note that we previously had another function called add that took two parameters as input. That function is still usable.

We can also return an object itself. If we call a function through the object d1, we can return d1 itself from that function.

return \*this;

C++

## Structures and Classes

Remember that in C++, we can actually use a structure to hold functions as well. However, this is discouraged. Also, in a class, by default, everything added is private, even if we do not use the private keyword. The only things that are public are those that come after the public keyword. In a structure on the other hand, everything is public by default.

## Classes, Objects and Memory

Remember that a class definition is simply information about what data an object of that class will have and what functions will be related to that object. It in itself does not hold any memory space.

When we created an object, the required memory space for those pieces of data or variables is allocated to that object. If we create another object, another distinct memory space will be allocated to it.

However, there are not multiple copies of the functions associated with the class, only one. When an object of that class calls the functions, they are executed and are given memory to operate. Once they are terminated, the memory space they took is freed up again. Essentially, data is not shared between objects, but functions are.

## Static Class Data

The only exception to the rule that separate objects have separate data and that the class definition does not occupy any memory is the static variable. These are shared amongst all the objects of that class and are allocated memory space as soon as the class definition is given and the variable is initialized. To use the static variable, we *have* to initialize it. After that, all objects have access to that single variable. Regardless of which object edits that variable, the results will be seen across all objects. Note that we do not actually need to create any objects to first create the static variable. However, the variable will only be accessible through the objects. (Also, apparently static variables have to public all the time. Taufiq said this.)

#include<iostream>  
using namespace std;  
  
class StVar  
{  
 static int stVar; *//look at us ignoring Taufiq*public:  
 void setVar(int a)  
 {  
 stVar = a;  
 }  
 int getVar()  
 {  
 return stVar;  
 }  
  
};  
int StVar :: stVar = 0; *//initialization*int main()  
{  
 StVar s1, s2;  
 cout<<s1.getVar()<<endl;  
 s1.setVar(5);  
 cout<<s2.getVar();  
}

C++

The program above creates two objects of the StVar class. When the value of stVar is printed from the first object, it gives 0. Then, we set the value of stVar from the first object. When we print the value of stVar from the second object, it will show the changed value.

Notice that we are setting a value for stVar from outside the class, even though it is a private variable. That is not actually what is happening there. This is actually the initialization of a static variable, similar to how constructor functions initialized values. We are only allowed to do this one time. stVar is genuinely private and we cannot access it from outside the class.

## Static Functions

A static function can be called even without initializing an object. It cannot access any variables that are not static variables.

Class Something  
{  
 static int a;  
 int b;  
public:  
 static void someFunction()  
 {  
 cout<<a;  
 *//cout<<b;* }  
}

C++

The commented-out line would cause an error.

## Constant Functions

A const function is declared like this:

void display() const;

C++

Such a function will not allow us to edit any values inside them. We only have read access. A display function is perfect in this sort of scenario since we do not want to be able to edit anything, even by mistake.

A const function does not allow the use of any functions inside it that are not const functions as well. This is to prevent values from being edited through those functions.

We can only use const functions inside classes, not with normal functions.

## Constant Objects

const objects do not allow their values to be edited. We can only call const functions on them. For example:

const Distance d3 = d2.add(d1);

C++

This works fine, given that add() is declared as a const function. However, this causes an error:

d3.setDistance(2, 2);

C++