

2021/2022



University of Technology - IRAQ
Computer Sciences Department

Object-Oriented Programming in C++













2nd Level
1st Course

Asst. Lect. Saif Bashar

Lecture notes about the basic concepts of object-oriented programming with C++ for more information and resources visit my website at: s4ifbn.com



Functions Review

-  5.1 Introduction
-  5.2 Function Implementation
-  5.3 Function Calling
-  5.4 Function Declaration
-  5.5 Functions' Types
-  5.6 Passing Parameters
-  5.7 Function Overloading
-  5.8 Default Arguments
-  5.9 Global & Local Variables
-  Exercises 5



5.1 Introduction

Another fundamental concept in all of programming languages is the concept of a function, we have seen a single function so far in this book which is the **main** function, in this chapter we will see how to build our own functions and how to make them run whenever we want. Functions give us more organized, readable and efficient programs.

Functions main advantage is to group pieces of code into a module and run it whenever the program needs, for example if we have a program that calculate the factorial for the number 5, and we used a loop as we did before in the previous chapter, now suppose later on the same program we needed to calculate the factorial for the numbers 6,7,8. We have to write a loop for each number. So, we have a long program with four loops all do the same job which is calculating a factorial for a number.

We can use a function that we will build with only one loop, then pass to it the numbers we want to calculate the factorial for and the function will return to us the desired output. This will result in a lot shorter, faster program and easier to fix and develop, the code with functions is more maintainable.

The concept of functions in programming is similar to the function concept of mathematics (sort of) in mathematics as we recall functions written in the following form:

$$y = f(x)$$

Which means that the function has a name **f** and it takes an input **x** and returns an output that will be stored in the variable **y**, so for example the trigonometric sine function if we give it 30 degrees as an angle it will return to us the trigonometric ratio for this angle which is 0.5

$$\begin{aligned} y &= \sin(30) \\ y &= 0.5 \end{aligned}$$





5.2 Function Implementation

The syntax of implementing a function in C/C++ require from us to think of many things:

- 1- **Function's name:** we have to think of a suitable name for our function, we have to give it an identifier with the same conditions that we applied on variables, see Chapter 2.
- 2- **Function's input:** what this function should take as an input? How many parameters or arguments should it receive? note that we can build a function that do not receive any input as we will see later.
- 3- **Function's body:** what is the code will be inside the curly braces? The code that will represent the function's task. What we want the function to do for us?
- 4- **Function's Output Type:** what is the output type that the function will produce, what will it return? Does it return an **int**, a **float**, a **string** or something else? Note if a function does not return anything we use the keyword **void**.



Function Implementation (Definition)

```
returnType functionName(arguments) {  
    statement1;  
    statement2;  
    statement3;  
    return value;  
} //end of function block
```



5.3 Function Calling

After implementing a function, it will not be executed. Executing functions happens only when we call them, calling a function happens when we write its name and give it proper arguments that match its implementation.



Function Calling

```
int main () {  
    int result = functionName (parameters) ;  
    return 0 ;  
}
```

Note that we can call a function from the **main** function or from any other user defined function.

As an example, I will try to write the factorial program that we saw in Chapter 4 – Program 4.2, using functions, the code will be something like Program 5.1.

You might say that a program with function (**Program 5.1**) took much more lines of code than a program without function (**Program 4.2**), I will tell you for calculating a single number you are right, but what if you needed to calculate factorial for more numbers? In this situation writing a function is the best practice to write code since you don't have to repeat the factorial code over and over again.



Program 5.1

```
1 // calculating factorial using function
2 #include<iostream>
3 using namespace std;
4
5 long fact(int number){
6     long result = 1;
7     for (int i=number; i>1; --i){
8         result *= i;
9     }
10    return result;
11 } //end of fact function
12
13 int main () {
14     int number;
15     long result;
16
17     cout<<"Give me a Number: ";
18     cin>>number;
19
20     result = fact(number);
21     cout<<"The Result: "<<result;
22
23     return 0;
24 } //end of main function
```

You can now calculate the factorial for as many numbers as you want with only one function implementation, but you have to call the function with the number you want to calculate its factorial, something like this:

```
int main () {

    cout<<fact (3)<<endl;
    cout<<fact (4)<<endl;
    cout<<fact (5)<<endl;
    cout<<fact (6)<<endl;
    return 0;

}
```



Note that in **Program 5.1** in **Line 13** I specified the return type for the **main** function as of type **int** and in **Line 23** I added the **return 0;** statement as the **main** function will return the integer zero to the operating system this will indicate that our program executed and finished successfully.



Matching types

the function's returned value must match the return type specified in the function's implementation, or you will encounter errors.

5.4 Function Declaration

Function declaration sometimes also called function's **prototype** or function's **signature** it is the statement that declare the function to the program.

As we saw in the previous program we wrote the function above the **main** function, this is not mandatory, we can write the function below the **main** function but in this case, we have to add the function's declaration statement.

The reason behind why we need to declare a function if its written below the **main** function, is we know that the compiler read the program from top to bottom, and if the function's implementation was below the **main** function and the function call was inside the **main** function, so the compiler will read the function call before its implementation. Which means that the compiler does not know anything about the function we are calling. So, we need to include a declaration before we call a function, for the compiler to understand what you are calling.



Program 5.1 can also be written like this:



Program 5.2

```
1 // calculating factorial using function
2 #include<iostream>
3 using namespace std;
4
5 int fact(int);
6
7 int main () {
8     int number;
9     cout<<"Give me a Number: ";
10    cin>>number;
11    cout<<"The Result: "<< fact(number);
12    return 0;
13 } //end of main function
14
15 long fact(int number){
16     long result = 1;
17     for (int i=number; i>1; --i){
18         result *= i;
19     }
20     return result;
21 } //end of fact function
```

The factorial function's implementation below the **main** function and the prototype added in **Line 5**.

But however, there is a performance related issue here, when you put a function's implementation above the **main** function it is automatically considered as **inline function**, inline functions are faster to run than regular functions, inline functions save the time required to retrieve a function at runtime. You can also add the **inline** keyword before the function's return type, to make a function an inline function, this is used for short functions that the program frequently calls.



The function's prototype contains the function name and its return type and the number of arguments it takes and their types and a semi colon added at the end of the declaration statement.



Function Declaration

```
returnType functionName (argumentType) ;
```



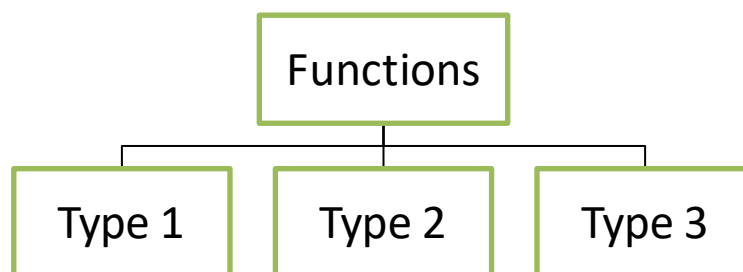
Video lecture: Functions

<https://youtu.be/L-5cmL-Edxk>

You can also put the prototype inside the **main** function, but it must be before the function call.

5.5 Functions' Types

We can classify functions according on what they receive as an input and what they return or not return as an output, as the following diagram describes functions types:



Type 1: when the function takes no parameters and don't return any value, we specify **void** as a return type, we can also put **void** in its arguments or leave the parenthesis empty, usually functions that don't return any value to the function call, they print out the output.





Let us write the factorial program again with a function of this type



Program 5.3

```
1 // calculating factorial using type 1 function
2 #include<iostream>
3 using namespace std;
4
5 inline void fact(void) {
6     int number;
7     long result = 1;
8     cout<<"Give me a Number: ";
9     cin>>number;
10    for(int i=number; i>1; --i){
11        result *= i; }
12    cout<<"The Result: "<<result;
13 }
14
15 int main () {
16     fact();
17     return 0; }
```

Type 2: when the function takes parameters and don't return any value, we specify **void** as a return type, usually functions that don't return any value to the function call, they print out the output.



The factorial program with a function of this type



Program 5.4

```
1 // calculating factorial using type 2 function
2 #include<iostream>
3 using namespace std;
4
5 inline void fact (int number) {
6     long result = 1;
7     for(int i=number; i>1; --i){
8         result *= i; }
9     cout<<"The Result: "<<result;
10 }
11
12 int main () {
13     int number;
14     cout<<"Give me a Number: ";
15     cin>>number;
16     fact (number);
17     return 0; }
```

Note that the calling of type 1 and type 2 functions, we just mention the function name with its parameters in a single line, see **Line 16** on both programs Program 5.3 and Program 5.4.

Type 3: when the function takes parameters and return the output to the function call, this type is the most commonly used by programmers as we wrote the factorial program using this type see **Program 5.1** or **Program 5.2**.

Note that the calling of type 3 functions, the function come either in an expression as of Program 5.1 **Line 20** or in the **cout** statement as of Program 5.2 **Line 11**.



5.6 Passing Parameters

The input to a function of type 2 or type 3 is passed from the function call to the function's implementation. This process is called passing parameters.

When we call a function like this:

```
int main () {  
  
    int a=90, b=88, c=78;  
    float v = average (a, b, c);  
  
    cout<<v;  
  
    return 0;  
}
```

This means that we are passing a copy of the values of the three variables **a**, **b** and **c**. The values 90, 88 and 78 are copied and the copies are passed to the function's implementation, this process is called sending parameters **by value**.

The passed copies will be received by the function's implementation as regular variables. We should match the parameter's number and type, the function's implementation should look like this:

```
float average (int a, int b, int c) {  
  
    return (a+b+c) / 3.0;  
}
```

The variables in the function implementation will receive the copied values from the function call in the order they were sent. The **a**, **b**, **c** variables in the implementation are not the same variables in the main function although they hold the same values, we can also change the variables



names it won't affect the program, the following implementation will work too:

```
float average(int x, int y, int z) {  
    return (x+y+z) / 3.0;  
}
```

The variables in the main function are called the **actual parameters** while the variables in the function implementation are called the **formal parameters**.

Another method of passing parameters is when we pass their addresses, and this is called **passing by pointer**. We use pointers to store the addresses.

Here we are passing the addresses:

```
int main() {  
    int a=90, b=88, c=78;  
    float v = average(&a, &b, &c);  
    cout<<v;  
    return 0;  
}
```

Note that the **&** operator is called the **reference operator** that returns the address of a variable which is represented in hexa-decimal, don't confuse the unary address operator with the binary bitwise and operator that we discussed in Chapter 3.



Try to test the reference operator to see that it gives you the variable's address in hexa-decimal value:

```
cout <<&a << endl <<&b;
```

When we pass an address, we have to store it in a special variable that store only addresses, this special variable is called a **pointer** (more about pointers in Chapter 9), so the implementation of the function will be like this.

```
float average(int *a, int *b, int *c) {  
    return (*a + *b + *c) / 3.0; }  
}
```

A pointer indicated by the **dereference operator** *, don't confuse the unary dereference operator with the binary multiplication operator that we discussed in Chapter 2.

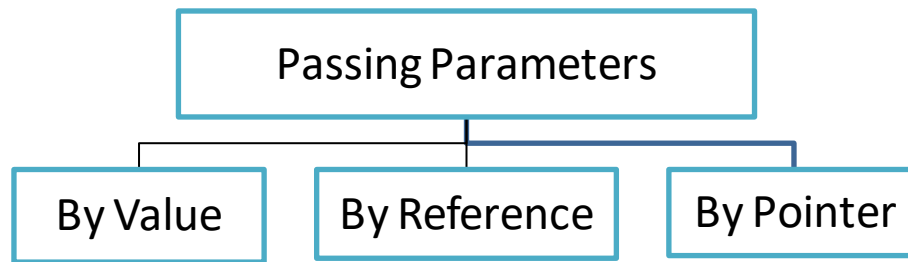
Here **a**, **b** and **c** are pointers that points to the original values of the variables in the **main** function.

Another easier way to pass addresses is called **passing by reference**

```
float average(int &a, int &b, int &c) {  
    return (a+b+c) / 3.0;  
}  
  
int main() {  
    int a=90, b=88, c=78;  
    float v = average(a, b, c);  
    cout<<v;  
    return 0;  
}
```



Three ways of passing parameters in C/C++



The difference between the first method (passing by value) and the other two methods (passing by reference and passing by pointer) is that in the first method when you pass a parameter by value this means the original value will remain the same, any changes on the passed value will affect only the copied value not the original, let me explain this with an example that swaps two values.



Program 5.5

```
1 // swap function pass by value
2 #include<iostream>
3 using namespace std;
4
5 inline void swap(int a, int b) {
6     int temp = a;
7     a = b;
8     b = temp;
9 }
10
11 int main () {
12     int a=33, b=66;
13     swap(a, b);
14     cout<<"Numbers after swapping: "<<endl;
15     cout<<"a = "<<a<<endl<<"b = "<<b;
16     return 0; }
```



The output will be:

```
Numbers after swapping:  
a = 33  
b = 66
```

Note that the original values did not swap because we were swapping the copies not the original values.

Passing values by reference or by pointer do the opposite, the change is done on the function will affect the original values in the program, look at Program 5.6 and Program 5.7.



Program 5.6

```
1  // swap function pass by pointer  
2  #include<iostream>  
3  using namespace std;  
4  
5  inline void swap (int *a, int *b) {  
6      int temp = *a;  
7      *a = *b;  
8      *b = temp;  
9  }  
10  
11  int main () {  
12      int a=33, b=66;  
13      swap (&a, &b);  
14      cout<<"Numbers after swapping: "<<endl;  
15      cout<<"a = "<<a<<endl<<"b = "<<b;  
16      return 0; }
```




The output will be:

```
Numbers after swapping:
a = 66
b = 33
```

You can see that the swap function affected and swapped the original values in the **main** function.



Program 5.7

```
1 // swap function pass by reference
2 #include<iostream>
3 using namespace std;
4
5 inline void swap (int &a, int &b) {
6     int temp = a;
7     a = b;
8     b = temp;
9 }
10
11 int main () {
12     int a=33, b=66;
13     swap (a, b);
14     cout<<"Numbers after swapping: "<<endl;
15     cout<<"a = "<<a<<endl<<"b = "<<b;
16     return 0; }
```

The output will be:

```
Numbers after swapping:
a = 66
b = 33
```



So, which way is better? Definitely passing by reference and by pointer is better because it results in a faster program with better performance especially when passing large files, passing by value will slow down your program because it involves coping data and passing them to functions.

Although sometimes you don't have large chunks of data and you don't want your original values to change then passing by value is the correct choice.

5.7 Function Overloading

The word overloading in programming world means give something more than one job, function overloading means one function name and multiple implementations, the correct implementation is executed according to the number of parameters passed or the return type of a function.

Let me explain this with an example, let's build a function that calculate a volume for a cube and a cylinder. One solution is to write two functions with different names, the first function to calculate the volume of a cube and the other one to calculate the volume of the cylinder.

Another solution you can benefit from the feature of function overloading, by writing two functions with the same name but different parameters.

Program 5.8 below has two functions with the same name but different number of parameters, this is called function overloading, the function that calculate the cube's volume will be executed when we call it and pass to it only one parameter, see **Line 18**.

The other volume function that calculates the cylinder volume will be executed when called and passed to it two parameters see **Line 25**.



Note that function overloading can happen not only if the number of arguments is different, it can happen also with the same number of arguments but different data types, or even different return type, the idea is the name of the function is fixed but its signatures vary.



Program 5.8

```
1 // function overloading
2 #include<iostream>
3 using namespace std;
4
5 int volume(int a){
6     return (a * a * a);
7 }
8
9 double volume(int h, int r){
10     const float pi = 3.14;
11     return (pi * r * r * h);
12 }
13
14 int main () {
15     int length, height, radius;
16     cout<<"Give me the cube length: ";
17     cin>>length;
18     cout<<"Cube volume: "<<volume(length)<<endl;
19
20     cout<<"Give me the cylinder height: ";
21     cin>>height;
22     cout<<"Give me the cylinder radius: ";
23     cin>>radius;
24     cout<<"Cylinder volume: ";
25     cout<<volume(height, radius);
26
27     return 0;
28 }
```



Video lecture: Important Notes on Functions

https://youtu.be/L6_4yx4Nx-s



5.8 Default Arguments

We can specify default arguments for a function in the function's implementation, when we specify the default argument it will be assigned to the parameter if we don't pass one to the function, for example, if we have a function that print a message, implemented as follows:

```
void print(string msg="Hello from default argument"){  
    cout<<msg;  
}
```

Now if we call it and pass a parameter to it like this:

```
string msg = "Hello from main function";  
print(msg);
```

The output will be:

```
Hello from main function
```

But when we call it and don't pass anything:

```
print();
```

The output will be:

```
Hello from default argument
```

The default argument will give the value to the **msg** variable.



Video lecture: Default Arguments

<https://youtu.be/LPwvSEfHk8M>



5.9 Global & Local Variables

We learned about variables in Chapter 1 and we saw there are a lot of things related to variables, a variable has name, value, address, size, range and a data type. We also learned that a variable has a storage specifier.

Now I want to tell that a variable also has a life or a **scope**, the variable is born or created when you declare it, and dies when its scope ends, for example if you declare a variable in **main** function it will be created in memory and dies when the program reaches the **return 0;** statement.

We can create scopes inside other scopes using the curly braces, check the following program:



Program 5.9

```
1 // variable scope
2 #include<iostream>
3 using namespace std;
4
5 int main () { // start of main function scope
6     int var = 10;
7
8     { // new internal scope
9         int var = 50;
10        cout<< var <<endl;
11    } // end of internal scope
12
13    cout<< var <<endl;
14
15    return 0;
16 }
```



Program 5.9 has two variables named **var**, you might think they are the same but they are not, they have the same name but different scopes, they two separate variables.

The variable **var** created at **Line 6** will die at **Line 15**, and the **cout** in **Line 13** will print its value which is 10.

While the second variable **var** is another variable stored in different location in memory created at a new scope in **Line 9** and died at **Line 11** when its scope ended, so the **cout** in **Line 10** will print 50 the value of **var** in this scope, not the value of the other **var** in the outer scope.

We have seen that functions contain curly braces this mean every function have a scope of its own, when a function is called and parameters are passed, the functions variables are created in memory and deleted when the function's scope ends. The variables inside a function's scope are called **local variables**, and can be accessed only inside the function.

We can define **global variables** above the **main** function and they can be accessed anywhere in the program and they die when the program ends, see the following program:



Program 5.10

```
1 // global variables
2 #include<iostream>
3 using namespace std;
4
5 string name = "saif";
6
7 int main () {
8     cout<<"Program written by: "<< name;
9     return 0;
10 }
```



Program 5.10 has a global variable **name** declared top of the **main** function and printed inside the **main** function, it also can be accessed from any other function in the program.

Note that if you have two global variables of the same name in two different files, then you included one file into the other file, you will have duplicate variables with the same name. In this scenario we can use the **extern** keyword to refer to the variable in the included file.

Chapter 2 also introduced you to the **auto** keyword as the default state of a variable, if we create a variable inside a function it is called a local variable and an **automatic variable**, it will be created every time we call the function and deleted when the function ends at each call.

There is another type of variables that are created at the first function call only and live as long as the program is running these are called **static variables**, we can create a static variable by using the keyword **static** in its declaration.



Program 5.11

```
1 // static vs automatic
2 #include<iostream>
3 using namespace std;
4
5 void Myfunc () {
6     int a=10;           //automatic variable
7     a++;
8     cout<<"a= "<<a<<endl;
9     static int b=20;    //static variable
10    b++;
11    cout<<"b= "<<b<<endl; }
12
13 int main () {
14     Myfunc ();
15     Myfunc ();
16     Myfunc ();
17     return 0; }
```



The **main** function in Program 5.10 has three function calls to a function called **Myfunc()**, this function has two variables an automatic variable **a** that will be created and deleted three times and a static variable **b** that will be created once at the first call and deleted once when the program ends.

The output of Program 5.11

```
a= 11
b= 21
a= 11
b= 22
a= 11
b= 23
```

Don't confuse **const** with **static**, because **const** mean the value is constant and cannot be change, while **static** means the variable's existence is not changing while we can change its value.



Video lecture: More about Variables

<https://youtu.be/lHtgEwR65c8>



Exercises 5

Q1: Write a program that solve the following math series using functions

$$y = 1 - \frac{1}{2!} + \frac{1}{3!} - \frac{1}{4!} + \dots \dots \frac{1}{n!}$$



Video lecture: Solution for Q1 - Exercises 5

<https://youtu.be/-ldNgzepkL4>

Q2: Write a program using function that count uppercase letters, in 20 letters entered by the user in the main function



Video lecture: Solution for Q2 - Exercises 5

https://youtu.be/dQEGcsoV9_I

Q3: Write a program using function that reads two integers (feet and inches) representing distance, then convert this distance to meters.

1 foot = 12 inch

1 inch = 2.54 cm



Video lecture: Solution for Q3 - Exercises 5

<https://youtu.be/XxQayunYdlU>

Q4: Write a program using function that reads an integer (T) representing time in seconds, then convert it to the equivalent hours, minutes and seconds in this form: Hour : Minute : Second



Video lecture: Solution for Q4- Exercises 5

<https://youtu.be/PD2tMCSFg1o>

Q5: Write a program using function to see if a number is an integer (odd or even) or not an integer



Video lecture: Solution for Q5- Exercises 5

<https://youtu.be/m0NYp2CPQm8>



Q6: Write a program using function to input student average and return:

- 4 if the average between 90-100
- 3 if the average between 80-89
- 2 if the average between 70-79
- 1 if the average between 60-69
- 0 if the average is lower than 60



Video lecture: Solution for Q6- Exercises 5

<https://youtu.be/K08iwf2TWuY>

Q7: Write a program using function to find the permutation of n



Video lecture: Solution for Q7- Exercises 5

https://youtu.be/b01jTq_uk7U

Q8: Write a program using function to find the nth Fibonacci number



Video lecture: Solution for Q8- Exercises 5

<https://youtu.be/pqBiu4CSClQ>

Q9: Write a program using function to calculate the factorial of an integer entered by the user in the main function



Video lecture: Solution for Q9- Exercises 5

<https://youtu.be/CtdrgpRZIQE>

Q10: Write a program using function to evaluate the following equation

$$Z = \frac{x! - y!}{(x - y)!}$$



Video lecture: Solution for Q10- Exercises 5

<https://youtu.be/Rlh9WD7qFHc>



Q11: Write a program using function to test the year if it is a leap year or not



Video lecture: Solution for Q11- Exercises 5

https://youtu.be/77V_xoV93SQ

Q12: Write a program using function to find x^y



Video lecture: Solution for Q12- Exercises 5

<https://youtu.be/fk9urKJrj6Q>

Q13: Write a program using function to reverse an integer number



Video lecture: Solution for Q13- Exercises 5

<https://youtu.be/o3o2gGvqKeA>

Q14: Write a program using function to convert any character from capital to small or from small to capital



Video lecture: Solution for Q14- Exercises 5

<https://youtu.be/pPV5mmu9LzQ>

Q15: Write a program using recursive function to find power of n numbers











Video lecture: Solution for Q15- Exercises 5

<https://youtu.be/VbnKWF32MWo>



Classes & Objects

-  11.1 Overview on OOP
-  11.2 Defining a Class
-  11.3 Constructors
-  11.4 Constructor Overloading
-  11.5 Destructors
-  11.6 Scope Resolution Operator
-  11.7 Constant & Static Members
-  Exercises 11



11.1 Overview on OOP

This part of the book is dedicated to the Object-oriented features that were added to the original C language by Bjarne Stroustrup, in order to understand this chapter and the following chapters you must have a solid understanding on all the chapters discussed in part one, you have to gain mastery on all the fundamental concepts and control structures of the C language like conditions, loops, functions, arrays, strings, structures and pointers and all the details explained earlier.

So, what is OOP? first of all OOP is not a programming language, it is a style of programming that is applied to many programming languages, this mean that C++ is not the only object-oriented language in a matter of fact the first object oriented language was a language called **Simula** and what Bjarne did is that he applied the OOP features to the original C language and called the new version **C with classes** then the name changed to **C++**.

The old style of programming that we learned in the previous chapters called procedural programming, the program was consisting of a number of functions (procedures) that perform a specific task, and a **main** function that control the program execution, in OOP style of programming we have **classes** and **objects**, a class is a general term like the term *Bird* and a specific *dove* is an object created from the *bird* class, a *sparrow* is another object that we can create from the *bird* class, we cannot create a *cat* object from the *bird* class because it's simply doesn't make sense, we should make a class for cats to create cat objects, so objects belonging to a class share common attributes and functionalities, attributes also called **data members**, functions that belong to a class are called **member functions** or **methods**.

We create objects from classes, objects have attributes (data) and methods

Object = Attributes + Methods



Let me give you another example, the football player can be considered a class and the individual players are objects created from that class. Football players all share common attributes all of them have speed, length, skills, position, shoot strength ...etc. but each individual player has his own values, and all of them have functionalities they all run, pass, shot some of them attack some of them defend ...etc.



Figure 11.1 Football Player Class and it's Objects

I'm sorry that Ronaldo moved to Juventus and ruined this example for me. But you get the idea.

Anything in the world can be an object, an object is a specific item that belong to a class, it also called an **instance** of a class.

Classes and objects are similar to **structures** in C language (discussed in Chapter 8) we create objects from the class similar to what we did with structures when we created structure instances form it, the difference between a structure and a class is structures members are public while



class members are private by default, plus all the features of OOP that we will study are provided for the classes and objects.

So, why we need to program in the OOP style? The invention of this style of programming came to solve issues and problems that appeared in the old style of programming especially with large programs, programmers needed a way to protect their data from the rest of the program they needed a cleaner way to organize the code in classes that hold objects with common attributes and functionalities, this isolation of data or data hiding or **encapsulation** provided a layer of protection to the data. Classes also provide **data abstraction** which is the process of paying attention to important properties while ignoring the details of implementation, when you as a programmer don't need to know the details of a specific class you just want to benefit from its functionality all the implementation details are hidden inside the class. A class is like a capsule that contains data and methods.

Also, this grouping of code made error handling easier since the code of each class is isolated from the other classes in the program.

Another powerful feature of OOP is **inheritance** which simply means inherit the code that was previously written rather than write it all over again, this will save development time and make C++ programmers more productive, another feature is **polymorphism** which means a specific function can take many forms, the modules will change the way they operate depending on the context as we will see later. And other many new exiting features added to the programming languages to make programming more productive, safer, and easier to handle errors. Anywhere you go now (web development, mobile apps, desktop programs ... etc.), any famous language you pick you will see the features of OOP are supported and heavily used, we will study these features in details in the upcoming chapters.



To summarize, OOP support all the procedural programming features and also provide:

- Data abstraction
- Encapsulation (data hiding)
- Inheritance (reusability)
- Operator overloading
- Polymorphism
- Exception handling

Encapsulation, inheritance and polymorphism are considered the pillars of OOP paradigm

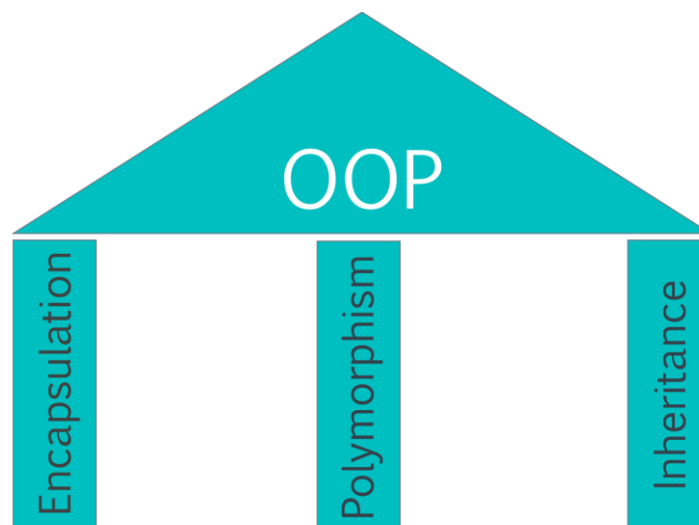


Figure 11.2 Pillars of OOP



11.2 Defining a class

Class definition depends on our understanding to the problem we want to solve, we have to think about what attributes or data we need to put in the class then what methods are needed to process that data, the syntax of defining a class in C++.



Defining a Class

```
class className {  
  
    private:  
        data1;  
        function1 () {}  
  
    public:  
        data2;  
        function2 () {}  
  
    protected:  
        data3;  
        function3 () {}  
};
```

The class definition will contain attributes or data and methods, with three access specifiers: **private**, **public** and **protected**

private: mean the data are private and can be accessed only from inside the class and some friends (more about friends in Chapter 12), note that this is the default access specifier for classes, I mean if you don't write an access specifier it will be considered private to ensure the encapsulation of data.



public: the data and methods are public and can be accessed from anywhere in the program, usually methods are specified public so we can call them from the **main** function, this section is also called the **class interface**.

protected: I will explain it in Chapter 15 when we will talk about inheritance.

So, for example if we want to find the area and circumference of any rectangle in the OOP style, first of all we need to think about what should we name the class, well in this case it's obvious let us call it **rectangle**, then we need to think what attributes or data this class will hold, as long as our problem is to find the area and the circumference so we need the dimensions **width** and **length**, so we can write the definition of the class as follows: (this definition will go above the **main** function)

```
class rectangle {  
private:  
    int width, length;  
};
```

Then we need to think about the methods, it is also obvious we need to calculate the area and the circumference so we need two methods for now, our definition will be after adding the two methods:

```
class rectangle {  
private:  
    int width, length;  
public:  
    int area() {  
        return width * length;  
    }  
    int circumference() {  
        return (width + length)*2;  
    }  
};
```



Note that the methods are added in the **public** section so we can call them from anywhere, and the data are in the **private** section to ensure encapsulation.

After completing the class definition, we need to create objects from this class, remember that the rectangle class will represent every rectangle in the world, there are infinite number of rectangles.

We declare objects anywhere in the program, let's declare an object in the **main** function

```
int main () {  
    rectangle r1, r2;  
  
    return 0;  
}
```

Note that we declared two objects **r1** and **r2**, they represent two rectangles, but now we have a problem, how we supposed to give the dimensions of each rectangle?

If we do this:

```
class rectangle {  
private:  
    int width=5, length=9;  
public:  
    int area() {  
        return width * length;  
    }  
  
    int circumference() {  
        return (width + length)*2;  
    }  
};
```



This will make all our rectangle objects have the same dimensions which is clearly wrong.

We can't do this also

```
int main () {  
    rectangle r1, r2;  
    r1.width = 5;  
    r1.length = 9;  
  
    r2.width = 2;  
    r2.length = 6;  
  
    return 0;  
}
```

We will get an error for this because **width** and **length** are private, we cannot access them from anywhere outside the class even from the **main** function.

We can make **width** and **length** public though but we will break the encapsulation paradigm.

The accepted solution for this problem i.e. initializing the data members of a class is to create a member function with this task, the function will be public and it will receive the values passed to it then assign these values to the data members of the class, the following program create a member function called **setValues()** that will initialize the data members for each object.



Note we call class members using the (dot) the member access operator like we used to do with structures. So, our rectangle program can be written like this:



Program 11.1

```
1 // Building a class & working with objects
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6 int width, length;
7 public:
8 void setValues (int x, int y) {
9 width = x;
10 length = y; }
11
12 int area () {
13 return width * length; }
14
15 int circumference () {
16 return (width + length)*2; }
17 };
18 int main () {
19 rectangle r1,r2;
20
21 r1.setValues (5,9);
22 r2.setValues (2,6);
23
24 cout<<"r1 area is: "<<r1.area()<<endl;
25 cout<<"r2 area is: "<<r2.area()<<endl;
26 cout<<"r1 circumference is: "
27 <<r1.circumference()<<endl;
28 cout<<"r2 circumference is: "
29 <<r2.circumference()<<endl;
30 return 0;
31 }
32 }
```



Video lecture: OOP Introduction

<https://youtu.be/i4SN8XFVpTk>



11.3 Constructors

There is a better way to initialize the class data members by using a special function called a **constructor**, the class constructor has these jobs:

- Initialize the class data members
- Reserve memory space for the object depending on its contents

A constructor has the following properties:

- It has the same name of the class
- It has no return type
- Called automatically when the object is declared
- Can be overloaded

Since the constructor is called automatically then we don't have to call it like we did with the **setValues()** function in Program 11.1, adding a constructor to our previous program see Program 11.2

The **setValues()** function was replaced by the class constructor on Line 8, see that the constructor has exactly the same name of the class, and it has no return type, and it initialize the class data.

The **main** function has the declaration of two objects, note that we send the initial values for each object in the object declaration statement on Line 19, since the constructors will be called automatically we don't have to call them in code like we did in the **setValues()** function see Program 11.1 Line 21 and Line 22.



Program 11.2

```
1 // Adding a constructor
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6     int width, length;
7 public:
8     rectangle(int x, int y){
9         width = x;
10        length = y; }
11
12     int area() {
13         return width * length; }
14
15     int circumference(){
16         return (width + length)*2; }
17 };
18 int main () {
19     rectangle r1 (5,9), r2 (2,6);
20
21     cout<<"r1 area is: "<<r1.area()<<endl;
22     cout<<"r2 area is: "<<r2.area()<<endl;
23     cout<<"r1 crcumference is: "
24         <<r1.circumference()<<endl;
25     cout<<"r2 crcumference is: "
26         <<r2.circumference()<<endl;
27     return 0;
28 }
```

We can also write the constructor like this:

```
rectangle(int x, int y): width (x), length (y) { }
```

This syntax is called **member initialization list**



11.4 Constructor Overloading

The overloading concept in programming means assign more than one job, we have seen function overloading in Chapter 5, function overloading means multiple implementations for the same function, the correct function is called depending on number or type of the arguments it receives. A class constructor is also a function and it can be overloaded, see the following program:



Program 11.3

```
1 // constructor overloading
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6     int width, length;
7 public:
8     rectangle(){
9         cout<<"Enter Dimensions: ";
10        cin>>width>>length; }
11
12    rectangle(int x, int y){
13        width = x;
14        length = y; }
15
16    int area(){
17        return width * length; }
18
19    int circumference(){
20        return (width + length)*2; }
21 };
22 int main () {
23     rectangle r1;
24     rectangle r2(2,6);
25     cout<<"r1 area is: "<<r1.area()<<endl;
26     cout<<"r2 area is: "<<r2.area()<<endl;
27     cout<<"r1 crcumference is: "
28         <<r1.circumference()<<endl;
29     cout<<"r2 crcumference is: "
30         <<r2.circumference()<<endl;
31     return 0; }
```




Note that the class in Program 11.3 have two constructors, the first one receives no parameters because it reads the rectangle dimensions, the other constructor receives two values and assign them to the class data.

The first constructor will be called automatically when the object **r1** is declared in Line 23, the second constructor will be called automatically when object **r2** is declares in Line 24.

We can add as many constructors as we want in the class definition, but remember that an object will call only one constructor at runtime.



Video lecture: Constructor Overloading

<https://youtu.be/MQsjwSxMbD4>

11.5 Destructors

The other special function that can be added to the class definition is the destructor. The destructor job is the opposite of constructor, it destructs the object to free the memory occupied by it.

Destructor job is:

- Remove the object and free up the memory that was reserved for it.

A destructor has the following properties:

- It has the same name of the class
- The name begins with tilde ~
- It has no return type
- It has no arguments
- Called automatically at the end of the object's scope, or when we delete the whole object
- Cannot be overloaded



So, the destructor job is to remove the object from memory, by removing the object we need to delete its data, but we cannot delete variables since they represent fixed memory locations, so we have to work with pointers, we can delete a pointer by making it point to something else so the location it was pointing to will be freed and returned to the operating system.

We delete pointers by using the **delete** operator, but to use the delete operator we have to create the pointer using the **new** operator, as a dynamic variable created it in the **heap**.

So, our class definition will initialize the data in the constructor using the **new** operator then delete the data using the **delete** operator in the destructor, see Program 11.4

When we work with OOP and want to build a complete class we have to think about these things:

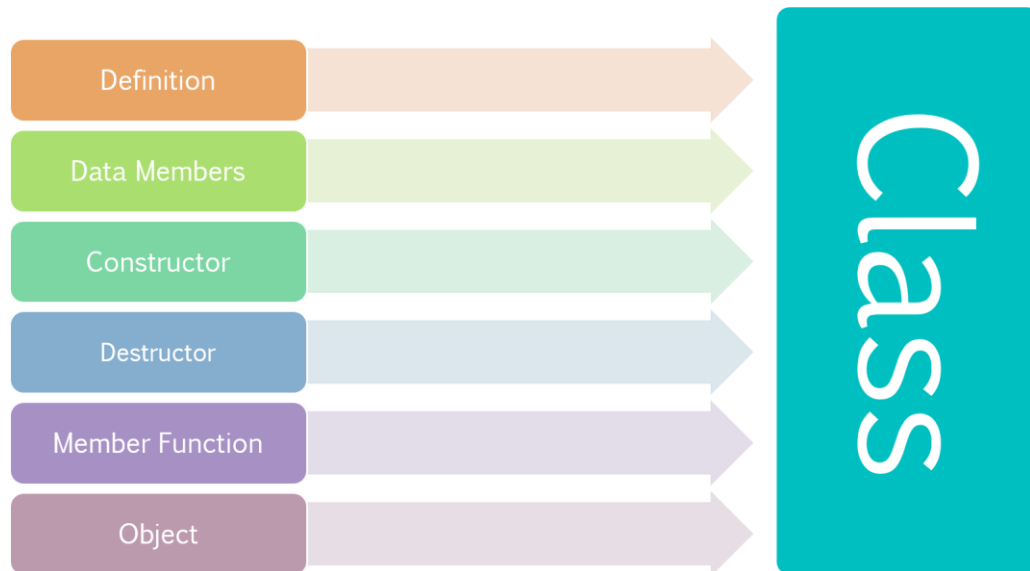


Figure 11.3 A Complete Class



Video lecture: Constructor & Destructor

<https://youtu.be/8xYuj94MVo4>



Program 11.4

```
1 // Adding a destructor
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6 int *width, *length;
7 public:
8     rectangle(int x, int y){
9         width = new int;
10        length = new int;
11        *width = x;
12        *length = y; }
13
14 ~rectangle(){
15     delete width;
16     delete length; }
17
18 int area(){
19     return *width * *length; }
20
21 int circumference(){
22     return (*width + *length)*2; }
23 };
24 int main () {
25     rectangle r1(5,9), r2(2,6);
26
27     cout<<"r1 area is: "<<r1.area()<<endl;
28     cout<<"r2 area is: "<<r2.area()<<endl;
29     cout<<"r1 crcumference is: "
30         <<r1.circumference()<<endl;
31     cout<<"r2 crcumference is: "
32         <<r2.circumference()<<endl;
33     return 0;
34 }
```



11.6 Scope Resolution Operator

We can separate the class definition from the implementation of the class member functions. I mean we can write the member functions of the class outside the class by using a special operator called the scope resolution operator written like this `::` as two adjacent colons, the syntax for using the scope resolution operator:



Scope resolution operator

```
returnType className::funcName () {  
}
```

Also, we need to leave the function prototype inside the class. Let us take the functions of the class in Program 11.2 outside the class definition, take a look at the following program:



Program 11.5

```
1 // Scope resolution operator  
2 #include<iostream>  
3 using namespace std;  
4 class rectangle {  
5     private:  
6     int width, length;  
7     public:  
8     rectangle(int, int);  
9     int area();  
10    int circumference();  
11 };  
12 rectangle::rectangle(int x, int y) {  
13     width = x;  
14     length = y; }  
15  
16 int rectangle::area() {  
17     return width * length; }  
18  
19 int rectangle::circumference() {  
20     return (width + length)*2; }
```



Note that we took out the constructor, the area and the circumference member function outside the class by using the scope resolution operator, and we left their prototypes inside the class.

The reason behind implementing the member functions outside the class is that in the case the class contains large number of functions it become hard to read and understand, taking out the member functions, make the class definition more condense and easier to read for the programmer.

Always remember the main six steps of building a complete class in C++

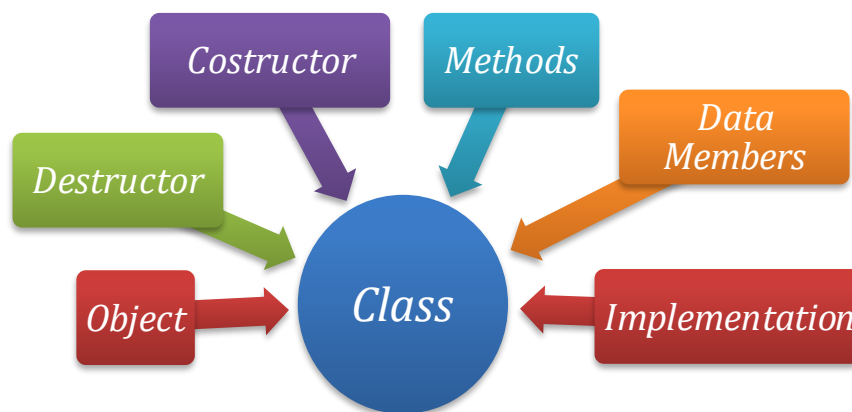


Figure 11.4 Steps of Building a class in C++

11.7 Constant & Static Members

We can use constant data members in the class definition, as you may recall we talked about constant variables in Chapter 2, we've seen that a constant variable means a variable that its value cannot be change.

We use the keyword **const** to declare constants and we know that constants must be initialized when declared, in classes when we declare a data member as constant we have to use with it **the member initialization list** method, see Program 11.6 that define a circle class with **PI** as constant data member.



Line 9 the constructor receives two values the integer **r** assigned to the **radius** data member and the **pi** assigned to the **PI** data member using the member initialization list syntax because it's a constant.



Program 11.6

```
1 // Constant data member
2 #include<iostream>
3 using namespace std;
4 class circle {
5 private:
6     int radius;
7     const float PI;
8 public:
9
10     static int total;
11
12     circle(int r):PI(3.14){
13         radius = r;
14         total++;
15     }
16
17     int area(){
18         return radius * radius * PI; }
19 };
20
21 int circle::total=0;
22
23 int main () {
24     circle c1(7), c2(4);
25     cout<<"c1 area is: "<<c1.area()<<endl;
26     cout<<"c2 area is: "<<c2.area()<<endl;
27     cout<<"you have "<<circle::total<<" circles";
28     return 0; }
```

We can set function's arguments as constants to protect them from altering, furthermore we can define constant objects with fixed values.



We can also have static data members declared inside the class; a static variable is created only once in the program so if we have multiple objects they will share the static member. We can count the number of objects created using static data members, or the number of functions calls and other useful purposes.

Static data members are initialized outside the class using the scope resolution operator.



Video lecture: Constant & Static Members

<https://youtu.be/3YAaeP9Kful>

11.8 Constant Objects

We can create objects as constant so we cannot change their data, for example if we have a class **Date** we can define its object as constant

```
const Date birthday(10, 10, 1990);
```

Note that only constructor and destructor will be able to modify the object, any other member function is not allowed to do so. Only constant methods can be called from a constant object, like the following method of class **Date**

```
int Date::getMonth() const{  
    return month;  
}
```

Objects that are not constant can call constant member functions and non-constant member functions.



Exercises 11

Q1: Write an object-oriented program in C++ that represent a simple arithmetic calculator



Video lecture: Solution for Q1 - Exercises 11

https://youtu.be/VWLFiLXw_fm



Friends



12.1 Friend Function



12.2 Friend Class



Exercises 12



12.1 Friend Function

We established in the previous chapter that the data members of a class should remain private, and by **private** access specifier we make the data accessible only for member functions of the class whether those functions were inside the class or implemented outside the class using the scope resolution operator.

There is a special function that can access private data of a class, this function is called **friend function**. Friend functions are regular functions that does not belong to the class, yet they can access the class's private data.

Implementing a friend function, require all the parts necessary to build a regular function plus keep in mind these two points:

- Use the keyword **friend** in the function's prototype inside the class definition
- Friend functions receive objects as arguments, in order to access the object's private data

Let's define a student class that contain the following data members:

- ID
- Name
- Age
- Deg1
- Deg2
- Deg3

Then implement a friend function that calculate the students average, check out Program 12.1



Program 12.1

```
1 // friend function
2 #include<iostream>
3 using namespace std;
4 class student {
5 private:
6     string Name;
7     int ID, Age, Deg1, Deg2, Deg3;
8 public:
9     student(int id, string n, int a, int d1, int
10 d2, int d3) {
11         ID=id;
12         Name=n;
13         Age=a;
14         Deg1=d1;
15         Deg2=d2;
16         Deg3=d3;
17     }
18     friend float average(student&);
19 };
20
21 float average(student &s){
22     float avg=(float) (s.Deg1+s.Deg2+s.Deg3)/3.0;
23     return avg;
24 }
25
26 int main(){
27     student s1(1003, "Mariam", 20, 76, 78, 98);
28     cout<<average(s1);
29     return 0;
30 }
```

Note that the **average()** function implemented outside the class as a regular function without using the scope resolution operator, but in order for it to be a friend function for the student class its prototype is added inside the class with the **friend** keyword Line 18.

Also note that the friend function received an object as an argument in order to access the student's degrees.



Friend functions are useful when we want them to process data for many objects belong to the same class or different classes.

Let's define an employee class that contain the following data members:

- Name
- Age
- Salary
- City

Use a friend function that find the summation of salaries of three employees

So, we will create the class with the data members above then create a constructor, and define three objects in the main function.

We will send these three objects by reference to a friend function **sum()** that will access the salary of each employee and return the summation value.

Checkout Program 12.2



Video lecture: Friend Function

<https://youtu.be/7YHg7ZNLNd8>



Program 12.2

```
1 // friend function for multiple objects
2 #include<iostream>
3 using namespace std;
4 class emp {
5 private:
6 string Name, City;
7 int Age, Salary;
8 public:
9 emp(string n, int a, int sal, string c) {
10     Name=n;
11     Age=a;
12     Salary=sal;
13     City=c;
14 }
15 friend int sum(emp&, emp&, emp&);
16 };
17
18 int sum(emp &e1, emp &e2, emp &e3) {
19     int s=e1.Salary+ e2.Salary + e3.Salary;
20     return s;
21 }
22
23 int main() {
24     emp e1("Mustafa", 42, 1500, "Baghdad");
25     emp e2("Ahmed", 39, 2500, "Babel");
26     emp e3("Muhanned", 40, 1200, "kut");
27     cout<<sum(e1,e2,e3);
28     return 0;
29 }
```



12.2 Friend Class

When we have several friend functions we can group them inside a class let's name it **class B**, and make class B a friend to other class let's call it **class A**, so we will need only to mention inside class A the prototype of the friend class B. And all its member functions will be considered friend functions to class A. you don't have to put the functions prototypes inside class A.

Steps of making class B friend to class A

- Put the class B declaration inside class A with the friend keyword
- When calling a member function of class B, pass to it an object from class A

Let us take a serious example that represent two classes that represent two banks namely AIRafidain and AIRasheed banks. Suppose a client of AIRasheed bank wants to transfer money from AIRashed bank to AIRafidain bank, let's make AIRasheed class a friend to the AIRafidain bank.



Video lecture: Friend Class

<https://youtu.be/jwcpnA32lyc>

The following example illustrate two bank classes the second class is a friend to the first class.



Program 12.3

```
1 // friend class for banks
2 #include<iostream>
3 using namespace std;
4 class AlRafidain {
5 private:
6 string Name;
7 int balance;
8 public:
9 AlRafidain(string n, int b) {
10     Name=n;
11     balance=b; }
12
13 float getBalance() {
14     return balance;
15 }
16
17 void deposit(float amount) {
18     balance+=amount;
19 }
20
21 void withdraw(float amount) {
22     balance-=amount;
23 }
24
25 friend class AlRashid;
26 }; //end of AlRafidain class
27
28 class AlRashid{
29 private:
30 string Name;
31 int balance;
32 public:
33 AlRashid(string n, int b) {
34     Name=n;
35     balance=b; }
36
37 float getBalance() {
38     return balance;
39 }
40
41 void deposit(float amount) {
42     balance+=amount; }
```



Program 12.3 cont.

```
43 void withdraw(float amount){
44     balance-=amount;
45 }
46
47 void transfer(AlRafidain& Client){
48     int amount;
49     cout<<"How much to transfer?: ";
50     cin>>amount;
51     if (amount<=Client. getBalance())
52     { balance+= amount;
53       Client.balance -= amount; }
54     else
55     cout<<"Not enough balance"<<endl;
56 }
57 }; //end of AlRashid class
58
59 int main () {
60     AlRafidain ahmed("Ahmed", 1000);
61     ahmed.deposit(500);
62     cout<<"Ahmed balance : ";
63     cout<<ahmed.getBalance ()<<"$"<<endl;
64
65     ahmed.withdraw(700);
66     cout<<"Ahmed balance now : ";
67     cout<<ahmed.getBalance ()<<"$"<<endl;
68
69     AlRashid ali("Ali", 500);
70     cout<<"Ali balance : ";
71     cout<<ali.getBalance ()<<"$"<<endl;
72
73     ali.transfer(ahmed);
74
75     cout<<"Ali balance : ";
76     cout<<ali.getBalance ()<<"$"<<endl;
77
78     cout<<"Ahmed balance : ";
79     cout<<ahmed.getBalance ()<<"$"<<endl;
80
81     return 0; }
```




Exercises 12

Q1: Write an object-oriented program in C++ that represent a car, the class constructor will set the values to all its data members, create a member function outside the class to display the car price, then create a friend function to increase the car price by 1000



Video lecture: Solution for Q1 - Exercises 12

<https://youtu.be/SvZqPijZhHk>

Q2: Write an object-oriented program in C++ that represent the student, the class constructor will set the values to all its data members, create another class to represent the university, use a member function of the second class to print the values of the first class








Video lecture: Solution for Q2 - Exercises 12

<https://youtu.be/zqjf197KQP8>



Pointers & Arrays of Objects

-  13.1 Array of Objects
-  13.2 Implicit Member Argument
-  13.3 Pointer to Object
-  13.4 Class Object Member
-  Exercises 13



13.1 Array of Objects

As we studied in Chapter 5, arrays can represent multiple elements of the same type, we worked with arrays of integers, arrays of floats, arrays of characters, arrays of strings and arrays of structures, now we will work with array of objects.

We can declare three or four objects in the main function, but when we have 100 object it is more efficient to declare an array of objects of size 100, the syntax is the same as array of structures.

As we always use loops in working with arrays, we can also send the whole array of objects to friend functions

Rewriting Program 12.1 using array of objects for 20 students

Note that in Line 34 a declaration of an array of size 20 this mean it will hold 20 objects of the class student, the important thing is that this declaration will invoke the constructor 20 times, a constructor for each object created, the constructor invoked will read the data from the user

```
student s[SIZE];
```

if the constructor was supposed to receive data from the main function, the array of object declaration should be like this. (if SIZE was 5)

```
student s[SIZE]={student(1,"Ali",20,56,78,98),  
                 student(2,"Ahmed",21,57,88,38),  
                 student(3,"Noor",21,57,80,56),  
                 student(4,"Sara",21,87,88,65),  
                 student(5,"Zahraa",21,70,82,81)  
};
```



Program 13.1

```
1 // Array of Objects
2 #include<iostream>
3 #define SIZE 20
4 using namespace std;
5 class student {
6 private:
7     string Name;
8     int ID, Age, Deg1, Deg2, Deg3;
9 public:
10     static int counter;
11     student () {
12         cout<<"Object Number "<<counter<<endl;
13         cout<<"Enter student ID: "; cin>>ID;
14         cout<<"Enter student Name: "; cin>>Name;
15         cout<<"Enter student Age: "; cin>>Age;
16         cout<<"Enter student Deg1: "; cin>>Deg1;
17         cout<<"Enter student Deg2: "; cin>>Deg2;
18         cout<<"Enter student Deg3: "; cin>>Deg3;
19         counter++;
20     }
21
22     string getName () {
23         return Name;
24     }
25
26     float average () {
27         return (Deg1 + Deg2 + Deg3)/3.0;
28     }
29 };
30
31 int student::counter=1;
32
33 int main () {
34     student s[SIZE];
35
36     for(int i=0; i<SIZE; i++)
37         cout<<s[i].getName()<<" Average "
38             <<s[i].average()<<endl;
39
40     return 0;
41 }
```



13.2 Implicit Member Argument

When a class member function is called there is an implicit argument passed to it, this argument represents the object itself we can refer to this object using **this** keyword.

this represent a pointer that points on the current object, since it's a pointer to object we have to use the arrow operator **->** to access the object's members.

Rewriting Program 11.2 using **this** pointer



Program 13.2

```
1 // using this pointer
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6 int width, length;
7 public:
8     rectangle(int width, int length) {
9         this->width = width;
10        this->length = length; }
11
12    int area() {
13        return width * length; }
14
15    int circumference() {
16        return (width + length)*2; }
17 };
18 int main () {
19     rectangle r1 (5,9), r2 (2,6);
20
21     cout<<"r1 area is: "<<r1.area()<<endl;
22     cout<<"r2 area is: "<<r2.area()<<endl;
23     cout<<"r1 crcumference is: "
24         <<r1.circumference()<<endl;
25     cout<<"r2 crcumference is: "
26         <<r2.circumference()<<endl;
27     return 0;
28 }
```



13.3 Pointer to Object

We can create pointers to objects in our programs and use the arrow operator to access the object's member, it's more efficient to use pointers to objects and it gives the programmer the control over the object's scope.

We declare a pointer to object like the following code (refer to Program 13.2)

```
rectangle *r1, *r2;
```

After declaring the pointers, we have to reserve a place in memory for them using the **new** operator like the following syntax:

```
r1 = new rectangle (5, 9);  
r2 = new rectangle (2, 6);
```

When we finish working with these objects we can delete them using the **delete** operator

```
delete r1;  
delete r2;
```

Using the delete operator will call the object destructor to free the memory occupied by the object.

Program 13.2 can be rewritten using pointer to object, see Program 13.3



Program 13.3

```
1 // pointer to object
2 #include<iostream>
3 using namespace std;
4 class rectangle {
5 private:
6 int width, length;
7 public:
8     rectangle(int width, int length) {
9         this->width = width;
10        this->length = length; }
11
12    int area() {
13        return width * length; }
14
15    int circumference() {
16        return (width + length)*2; }
17 };
18 int main () {
19     rectangle *r1,*r2;
20     r1 = new rectangle(5,9);
21     r2 = new rectangle(2,6);
22
23     cout<<"r1 area is: "<<r1->area()<<endl;
24     cout<<"r2 area is: "<<r2->area()<<endl;
25     cout<<"r1 crcumference is: "
26         <<r1->circumference()<<endl;
27     cout<<"r2 crcumference is: "
28         <<r2->circumference()<<endl;
29
30     delete r1;
31     delete r2;
32
33     return 0;
34 }
```



13.4 Class Object Member

Classes are user defined data types UDT we can use them to define members of other classes, for example we can define line class that have starting and ending points, by using point class as its data member see the following program



Program 13.4

```
1 // class object member to object
2 #include<iostream>
3 using namespace std;
4
5 class point {
6     private:
7         int x, y;
8
9     public:
10        point(int x, int y) {
11            this->x = x;
12            this->y = y; }
13 };
14
15 class line {
16     private:
17         point start, end;
18     public:
19         line(int x1, int y1, int x2, int y2) :
20             start(x1, y1), end(x2, y2) {
21             }
22 };
23
24 int main () {
25     line *myline;
26     myline = new line(5,4,8,9);
27
28     delete myline;
29
30     return 0;
31 }
32
33
```




Exercises 13

Q1: Write an object-oriented program in C++ that represent the rectangle class, define two data members, with constructor overloading, add a member function outside the class for printing, then define four rectangle objects, use friend function to find the biggest rectangle area.



Video lecture: Solution for Q1 - Exercises 13

<https://youtu.be/RCF1m62FfrE>

Q2: Write an object-oriented program in C++ to represent 20 students, the class constructor will set the values to all data members for each student, then calculate the average for each student using friend function.



Video lecture: Solution for Q2 - Exercises 13

<https://youtu.be/29NeWoONJlc>