# Module-2

# Functions, Classes and Objects

## Table of Contents

Functions in C++ (Function & operator Overloading,Inline functions,Default Arguments in functions). The ways of supporting function has changed and improved in C++ as compared to C.Most of these changes are simple and straightforward.However there are a few changes which require you to adopt a new way of thinking and organizing your program's code.Many of these requirements were driven by object oriented facilities of C++.As we go along you would realise that these facilities were invented to make C++ programs safer and more readable than their C equivalents.Let us know the various issues involved in C++ functions.

Function Prototypes

A function prototype is a declaration that defines both : the arguments passed to the function and the type of value returned by the function. If we were to call a function *fool( )* which receives a *float* and *int* as arguments and returns a double value its prototype would look like this:

```
double fool(float,int);
```

The C++ compiler uses the prototype to ensure that the types of the arguments you pass in a function call are same as those mentioned in the prototype.If there is a mismatch the compiler points out immediately.This is known as strong type checking,something which C lacks.

Without strong type checking,it is easier to pass illegal values to functions.For example,a non-prototyped function would allow you to pass an*int* to a pointer variable,or a *float* to a *long int.*The C++ compiler would immediately report such types of errors.In C++ prototypes do more than making sure that actual arguments (those used in calling function) and formal arguments (those used in called function) match in number,order and type.As we would see later,C++ internally generates names for functions,including the argument type information.This information is used when several functions have same names.

Remember that all functions in C++ must be prototyped.This means that every function must have its argument list declared,and the actual definition of a function must exactly match its prototype in the number,order and types of parameters.

```
//K & R style

double fool(a,b)
```

```
int a;   float b;
{
    //some code
}
//prototype-like style
double fool(int a,float b)
{
     //some code
}
```

## Function Overloading

Another significant addition made to the capabilities of functions in C++ is that of *function overloading*.With this facility you can have multiple functions with the same name,unlike C,where all the functions in a program must have unique names.

In C every function must have a unique name.This becomes annoying.For example,in C there are several functions that return the absolute value of a numeric argument.Since a unique name is required,there is a separate function for each numeric data type.Thus there are three different functions that return the absolute value of an argument:

```
int abs(int i);
long labs(long l);
double fabs(double d);
```

All these functions do the same thing,so it seems unnecessary to have three different function names.C++ overcomes this situation by allowing the programmer to create three different functions with the same name.This is called as *Function Overloading*.Now we can write the function abs as:

```
int abs(int ii);
long abs(long ll);
double abs(double dd);
```

How does the C++ compiler know which of the *abs( )s* should be called when a call is made? It decides from the type of the argument being passed during function call.For example,if an *int* is being passed the integer version of *abs( )* gets called,if a *double* is being passed then the double version of*abs( )* gets called and so on.That's quite logical,you would agree.

What if we make a call like,

```
ch=abs('A')
```

We have not declared *abs( )* function to handle a *char*.Hence the C++ compiler would report an error.If we want that this call should result into a call to the *int* version of *abs( )*,we must make use of typecasting during the call,as shown below:

```
ch=abs((int);A');
```

Default Arguments in Functions

In C if a function is defined to receive 2 arguments,whenever we call this function we have to pass 2 values to this function.If we pass one value then some garbage value is assumed for the last argument.As against this,functions in C++ have an ability to define values for arguments that are not passed when the function call is made.

Let us understand this with an example:

```
void box(int sr=1,int sc=1,int er=24,int ec=80);
void main( )
{
        clrscr( );
        box(10,20,22,70);          //Passing all 4 arguments
        box(10,20,15);             //Passing 3 arguments
        box(5,10);                 //Passing 2 arguments
        box( );                    //Passing no arguments
}
```

When we call the function *box( )* with 4 arguments the box is drawn with the arguments passed.However when we call it with 3 arguments the default value mentioned in the prototype of *box( )* is considered for the last argument.Likewise when we call the function with 2 arguments the default values for the last 2 arguments are considered.Finally,when we call it with no arguments, a box is drawn with all the default values mentioned in the prototype. Thus the default arguments are used if the calling function doesn't supply them when the function is called.

Note: If one argument is missing when the function is called, it is assumed to be the last argument.Thus,the missing arguments must be the trailing ones./you can leave out the last 3 arguments,but you cannot leave out the last but one and put in the last one.

Default arguments are useful in 2 cases:

a) While making a function call if you don't want to take the trouble of writing arguments which almost always has the same value.
b) They are also useful in such cases where,after having written a program we decide to increase the capabilities of a function by adding another argument.Using default arguments means that the existing function calls can continue to use old number of arguments,while new function calls can use more.
Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:
type name ( parameter1, parameter2, ...) { statements }
where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
#include <iostream>
```

```
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

Ans: `The result is 8`

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)
               ↑      ↑
z = addition (  5   ,   3  );
```

The parameters and arguments have a clear correspondence. Within the main function we called

to additionpassing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to functionaddition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to rthe result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:
*return* (r);

finalizes function addition, and returns the control back to the function that called it in the first place (in this case,main). At this moment the program follows its regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

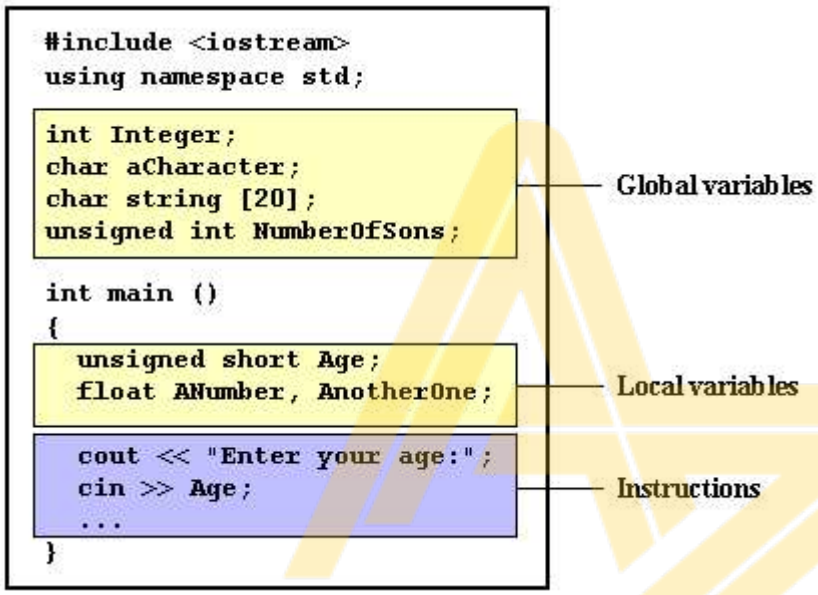
```
int addition (int a, int b)
 |8
z = addition (  5  ,  3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in main is:
cout << "The result is " << z;

That, as you may already expect, produces the printing of the result on the screen.

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables a, b or r directly in function main since they were variables local to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.




Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
 int r;
 r=a-b;
```

```
 return (r);
}

int main ()
{
 int x=5, y=3, z;
 z = subtraction (7,2);
 cout << "The first result is " << z << '\n';
 cout << "The second result is " << subtraction (7,2) << '\n';
 cout << "The third result is " << subtraction (x,y) << '\n';
 z= 4 + subtraction (x,y);
 cout << "The fourth result is " << z << '\n';
 return 0;
}
```

Ans:
```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

In this case we have created a function called subtraction. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function main we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

z = subtraction (7,2);

cout << "The first result is " << z;

If we replace the function call by the value it returns (i.e., 5), we would have:
z = 5;
cout << "The first result is " << z;

As well as
cout << "The second result is " << subtraction (7,2);

has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:
cout << "The second result is " << 5;

since 5 is the value returned by subtraction (7,2).

In the case of:
cout << "The third result is " << subtraction (x,y)

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:
z = 4 + subtraction (x,y);

we could have written:
z = subtraction (x,y) + 4;

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:
z = 4 + 2;
z = 2 + 4;

**Functions with no type. The use of void.**

If you remember the syntax of a function declaration:
type name ( argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
 cout << "I'm a function!";
}

int main ()
{
 printmessage ();
 return 0;
}
```

Ans: `I'm a function!`

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been

declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is: printmessage ();

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect: printmessage;

**Arguments passed by value and by reference.**

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function additionusing the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int addition (int a, int b)
                 ↑      ↑
z = addition (  5   ,   3  );
```

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and youtside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

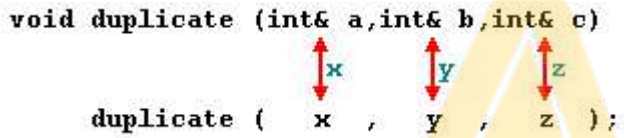
But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Ans: `x=2, y=6, z=14`

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.




To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on bwill affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:
*void* duplicate (*int*& a, *int*& b, *int*& c)

we had declared it this way:
*void* duplicate (*int* a, *int* b, *int* c)

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and zwithout having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
```

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

void prevnext (int x, int& prev, int& next)
{
 prev = x-1;
 next = x+1;
}
int main ()
{
 int x=100, y, z;
 prevnext (x, y, z);
 cout << "Previous=" << y << ", Next=" << z;
 return 0;
}
```

Ans: `Previous=99, Next=101`

**Default values in parameters.**

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
 int r;
 r=a/b;
```

```
 return (r);
}

int main ()
{
 cout << divide (12);
 cout << endl;
 cout << divide (20,4);
 return 0;
}
```

Ans: 6
5

As we can see in the body of the program there are two calls to function divide. In the first one: divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call: divide (20,4)

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

**Overloaded functions.**

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
// overloaded function
```

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

int operate (int a, int b)
{
 return (a*b);
}

float operate (float a, float b)
{
 return (a/b);
}

int main ()
{
 int x=5,y=2;
 float n=5.0,m=2.0;
 cout << operate (x,y);
 cout << "\n";
 cout << operate (n,m);
 cout << "\n";
 return 0;
}
```

Ans: `10`
`2.5`

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its

prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

**inline functions.**

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name ( arguments ... ) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

**Recursivity.**

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:
n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be:
5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
 if (a > 1)
  return (a * factorial (a-1));
 else
  return (1);
}

int main ()
{
 long number;
 cout << "Please type a number: ";
 cin >> number;
 cout << number << "! = " << factorial (number);
 return 0;
}
```

Ans:
```
Please type a number: 9
9! = 362880
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

**Declaring functions.**

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples. But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

type name ( argument_type1, argument_type2, ...);

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The

inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two int parameters with any of the following declarations:

*int* protofunction (*int* first, *int* second);

*int* protofunction (*int*, *int*);

Anyway, including a name for each variable makes the prototype more legible.

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int a);
void even (int a);

int main ()
{
  int i;
  do {
    cout << "Type a number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}

void even (int a)
```

```
{
 if ((a%2)==0) cout << "Number is even.\n";
 else odd (a);
}
```

Ans:
```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

A function is a group of statements that together perform a task. Every C++ program has at least one function which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

Defining a Function:

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
  body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

**Example:**

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers

int max(int num1, int num2)
{
  // local variable declaration
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;

}
```

Function Declarations:

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return_type function_name( parameter list );

For the above defined function max(), following is the function declaration:

int max(int num1, int num2);

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration: int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. For example:

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

```
// function declaration
int max(int num1, int num2);

int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;
  int ret;

  // calling a function to get max value.
  ret = max(a, b);

  cout << "Max value is : " << ret << endl;

  return 0;
}

// function returning the max between two numbers
int max(int num1, int num2)
{
  // local variable declaration
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
```

}

I kept max() function along with main() function and complied the source code. While running final executable, it would produce following result: Max value is : 200

## Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

| Call Type | Description |
|---|---|
| Call by value | This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| Call by reference | This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters:

When you define a function you can specify a default value for for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified this default value is ignored and the passed value is used instead. Consider the following example:

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

int sum(int a, int b=20)
{
  int result;

  result = a + b;

  return (result);
}

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;
```

```
  // calling a function again as follows.
  result = sum(a);
  cout << "Total value is :" << result << endl;

  return 0;

}
```

When the above code is compiled and executed, it produces following result:

Total value is :300

Total value is :120

**C++ function call by reference**

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int *x, int *y)
{
  int temp;
  temp = *x; /* save the value at address x */
  *x = *y; /* put y into x */
  *y = temp; /* put x into y */

  return;
```

```
}
```

To check the more detail about C++ pointers, kindly check C++ Pointers chapter.

For now, let us call the function **swap()** by passing values by reference as in the following example:

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

// function declaration
void swap(int *x, int *y);

int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;

  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;

  /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
  swap(&a, &b);

  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;

  return 0;
```

```
}
```

When the above code is put together in a file, compiled and executed, it produces following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200

After swap, value of b :100
```

## C++ function overloading programs

Function overloading in C++: C++ program for function overloading. Function overloading means two or more functions can have the same name but either the number of arguments or the data type of arguments has to be different, also note that return value has no role because function will return a value when it is called and at compile time we will not be able to determine which function to call. In the first example in our code we make two functions one for adding two integers and other for adding two floats but they have same name and in the second program we make two functions with identical names but pass them different number of arguments. Function overloading is also known as compile time polymorphism.

**C++ programming code**

```
#include <iostream>

using namespace std;

/* Function arguments are of different data type */

long add(long, long);
float add(float, float);

int main()
{
```

```
  long a, b, x;
  float c, d, y;

  cout << "Enter two integers\n";
  cin >> a >> b;

  x = add(a, b);

  cout << "Sum of integers: " << x << endl;

  cout << "Enter two floating point numbers\n";
  cin >> c >> d;

  y = add(c, d);

  cout << "Sum of floats: " << y << endl;

  return 0;
}

long add(long x, long y)
{
  long sum;

  sum = x + y;

  return sum;
}

float add(float x, float y)
{
```

```
  float sum;

  sum = x + y;

  return sum;
}
```

**C++ programming code for function overloading**

```
#include <iostream>

using namespace std;

/* Number of arguments are different */

void display(char []);  // print the string passed as argument
void display(char [], char []);

int main()
{
  char first[] = "C programming";
  char second[] = "C++ programming";

  display(first);
  display(first, second);

  return 0;
}

void display(char s[])
{
  cout << s << endl;
```

```
}

void display(char s[], char t[])
{
  cout << s << endl << t << endl;
}
```

Output of program:

C programming

C programming

C++ programming

**Virtual functions (C++ only)**

By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword `virtual` if you want the compiler to use dynamic binding for that specific function.

The following examples demonstrate the differences between static and dynamic binding. The first example demonstrates static binding:

The following is the output of the above example:Class A

When function `g()` is called, function `A::f()` is called, although the argument refers to an object of type `B`. At compile time, the compiler knows only that the argument of function `g()` will be a reference to an object derived from `A`; it cannot determine whether the argument will be a reference to an object of type `A` or type `B`. However, this can be determined at run time. The following example is the same as the previous example, except that `A::f()` is declared with the`virtual` keyword:

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

```
struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
  void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
  arg.f();
}

int main() {
  B x;
  g(x);
}
```

The following is the output of the above example:

**Class B**

The `virtual` keyword indicates to the compiler that it should choose the appropriate definition of `f()` not by the type of reference, but by the type of object that the reference refers to.

Therefore, a *virtual function* is a member function you may redefine for other derived classes, and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, even if you call that function with a pointer or reference to a base class of the object.

A class that declares or inherits a virtual function is called a *polymorphic class*.

You redefine a virtual member function, like any member function, in any derived class. Suppose you declare a virtual function named `f` in a class `A`, and you derive directly or indirectly from `A` a

class named `B`. If you declare a function named `f` in class `B` with the same name and same parameter list as `A::f`, then `B::f` is also virtual (regardless whether or not you declare `B::f` with the `virtual` keyword) and it *overrides* `A::f`. However, if the parameter lists of `A::f` and `B::f` are different, `A::f` and `B::f` are considered different, `B::f` does not override `A::f`, and `B::f` is not virtual (unless you have declared it with the `virtual` keyword). Instead `B::f` *hides* `A::f`. The following example demonstrates this:

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

struct A {
  virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
  void f(int) { cout << "Class B" << endl; }
};

struct C: B {
  void f() { cout << "Class C" << endl; }
};

int main() {
  B b; C c;
  A* pa1 = &b;
  A* pa2 = &c;
//  b.f();
  pa1->f();
  pa2->f();
}
```

The following is the output of the above example:

```
Class A
Class C
```

The function `B::f` is not virtual. It hides `A::f`. Thus the compiler will not allow the function call `b.f()`. The function `C::f` is virtual; it overrides `A::f` even though `A::f` is not visible in `C`.

If you declare a base class destructor as virtual, a derived class destructor will override that base class destructor, even though destructors are not inherited.

The return type of an overriding virtual function may differ from the return type of the overridden virtual function. This overriding function would then be called a*covariant virtual function*. Suppose that `B::f` overrides the virtual function `A::f`. The return types of `A::f` and `B::f` may differ if all the following conditions are met:

- The function B::f returns a reference or pointer to a class of type T, and A::f returns a pointer or a reference to an unambiguous direct or indirect base class of T.
- The const or volatile qualification of the pointer or reference returned by B::f has the same or less const or volatile qualification of the pointer or reference returned by A::f.
- The return type of B::f must be complete at the point of declaration of B::f, or it can be of type B.

The following example demonstrates this:

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

struct A { };

class B : private A {
  friend class D;
  friend class F;
};
```

```
A global_A;
B global_B;

struct C {
  virtual A* f() {
    cout << "A* C::f()" << endl;
    return &global_A;
  }
};

struct D : C {
  B* f() {
    cout << "B* D::f()" << endl;
    return &global_B;
  }
};

struct E;

struct F : C {

// Error:
// E is incomplete
// E* f();
};

struct G : C {

// Error:
// A is an inaccessible base class of B
// B* f();
```

```
};

int main() {
  D d;
  C* cp = &d;
  D* dp = &d;

  A* ap = cp->f();
  B* bp = dp->f();
};
```

The following is the output of the above example:

```
B* D::f()
B* D::f()
```

The statement `A* ap = cp->f()` calls `D::f()` and converts the pointer returned to type `A*`. The statement `B* bp = dp->f()` calls `D::f()` as well but does not convert the pointer returned; the type returned is `B*`. The compiler would not allow the declaration of the virtual function `F::f()` because `E` is not a complete class. The compiler would not allow the declaration of the virtual function `G::f()` because class `A` is not an accessible base class of `B` (unlike friend classes `D` and `F`, the definition of `B` does not give access to its members for class `G`).

A virtual function cannot be global or static because, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. You can declare a virtual function to be a friend of another class.

If a function is declared virtual in its base class, you can still access it directly using the scope resolution (`::`) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A base class containing one or more pure virtual member functions is called an *abstract class*.

## Operator Overloading

Operator Overloading is one of the most fascinating features of C++.It can transform,obscure program listings into intuitive obvious ones.By overloading operators we can give additional meaning to operators like +,*,-,<=,>=,etc. which by default are supposed to work only on standard data types like*ints*,*floats* etc. For example if *str1* and *str2* are two character arrays holding strings "Bombay" and "Nagpur" in them then to store "BombayNagpur" in a third string *str3*, in C we need to perform the following operations:

```
char str1[20] = "Nagpur";
char str2[ ] = "Bomaby";
char str3[20];
strcpy(str3,str1);
strcat(str3,str2);
```

No doubt this does the desired task but don't you think that the following form would have made more sense:

```
str3 = str1 + str2;
```

Such a form would obviously not work with C,since we are attempting to apply the + operator on non-standard data types (strings) for which addition operation is not defined. That's the place where C++ scores over C,because it permits the + operator to be overloaded such that it knows how to add

two strings.

Inline Functions

One of the important advantages of using functions is that they help us save memory space.As all the calls to the function cause the same code to be executed;the function body need not be duplicated in memory.

Imagine a situation where a small function is getting called several times in a program.As you must be aware,there are certain overheads involved while calling a function.Time has to be spent on passing values,passing control,returning value and returning control.In such situations to save the execution time you may instruct the C++ compiler to put the code in the function body directly inside the code in the calling program.That is, at each place where there is a function call in the source file, the actual code from the function would be inserted, instead of jump to the function. Such functions are called *inline functions*. The in-line nature of the individual copy of the function eliminates the function-calling overhead of a traditional function.The following program shows inline function at work.

```
#include<iostream.h>
inline void reporterror(char *str)
{
        cout<<endl<<str;
        exit(1);
}


void main( )
{
        //code to open source file
        if (fileopeningfailed)
        reporterrer("Unable to open source file");


      //code to open target file
```

```
    if(fileopeningfailed)
    reporterror("Unable to open target file");
}
```

**Function Pointers**

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function. But sometimes you would like to choose different behaviors at different times in essentially the same piece of code.

**Function Pointer Syntax**

The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you **understand** what's going on. Let's look at a simple example:

```
void (*foo)(int);
```

In this example, foo is a pointer to a function taking one argument, an integer, and that returns void. It's as if you're declaring a function called "*foo", which takes an int and returns void; now, if *foo is a function, then foo must be a pointer to a function. (Similarly, a declaration like int *x can be read as *x is an int, so x must be a pointer to an int.)

The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with (*func_name) where you'd normally just put func_name.

**Reading Function Pointer Declarations**

Sometimes people get confused when more stars are thrown in:

```
void *(*foo)(int *);
```

Here, the key is to read inside-out; notice that the innermost element of the expression is *foo, and that otherwise it looks like a normal function declaration. *foo should refer to a function that returns a void * and takes an int *. Consequently, foo is a pointer to just such a function.

**Initializing Function Pointers**

To initialize a function pointer, you must give it the address of a function in your program. The syntax is like any other variable:

```
#include <stdio.h>
void my_int_func(int x)
{
   printf( "%d\n", x );
}

int main()
{
   void (*foo)(int);
   /* the ampersand is actually optional */
   foo = &my_int_func;

   return 0;
}
```

(Note: all examples are written to be compatible with both C and C++.)

**Using a Function Pointer**

To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call. The act of calling it performs the dereference; there's no need to do it yourself:

```
#include <stdio.h>
void my_int_func(int x)
{
```

```
    printf( "%d\n", x );
}



int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

Note that function pointer syntax is flexible; it can either look like most other uses of pointers, with & and *, or you may omit that part of syntax. This is similar to how arrays are treated, where a bare array decays to a pointer, but you may also prefix the array with & to request its address.

**Function Pointers in the Wild**

Let's go back to the sorting example where I suggested using a function pointer to write a generic sorting routine where the exact order could be specified by the programmer calling the sorting function. It turns out that the C function qsort does just that.

From the Linux man pages, we have the following declaration for qsort (from stdlib.h):

```
void qsort(void *base, size_t nmemb, size_t size,
        int(*compar)(const void *, const void *));
```

Note the use of void*s to allow qsort to operate on any kind of data (in C++, you'd normally use templates for this task, but C++ also allows the use of void* pointers) because void* pointers can

point to anything. Because we don't know the size of the individual elements in a void* array, we must give qsort the number of elements, nmemb, of the array to be sorted, base, in addition to the standard requirement of giving the length, size, of the input.

But what we're really interested in is the compar argument to qsort: it's a function pointer that takes two void *s and returns an int. This allows anyone to specify how to sort the elements of the array base without having to write a specialized sorting algorithm. Note, also, that compar returns an int; the function pointed to should return -1 if the first argument is less than the second, 0 if they are equal, or 1 if the second is less than the first.

For instance, to sort an array of numbers in ascending order, we could write code like this:

```
#include <stdlib.h>

int int_sorter( const void *first_arg, const void *second_arg )
{
   int first = *(int*)first_arg;
   int second = *(int*)second_arg;
   if ( first < second )
   {
      return -1;
   }
   else if ( first == second )
   {
      return 0;
   }
   else
   {
      return 1;
   }
}
```

```
int main()
{
   int array[10];
   int i;
   /* fill array */
   for ( i = 0; i < 10; ++i )
   {
      array[ i ] = 10 - i;
   }
   qsort( array, 10 , sizeof( int ), int_sorter );
   for ( i = 0; i < 10; ++i )
   {
      printf ( "%d\n" ,array[ i ] );
   }

}
```

**Using Polymorphism and Virtual Functions Instead of Function Pointers (C++)**

You can often avoid the need for explicit function pointers by using virtual functions. For instance, you could write a sorting routine that takes a pointer to a class that provides a virtual function called compare:

```
class Sorter
{
   public:
   virtual int compare (const void *first, const void *second);
};

// cpp_qsort, a qsort using C++ features like virtual functions
void cpp_qsort(void *base, size_t nmemb, size_t size, Sorter *compar);
```

inside cpp_qsort, whenever a comparison is needed, compar->compare should be called. For classes that override this virtual function, the sort routine will get the new behavior of that function. For instance:

```
class AscendSorter : public Sorter
{

   virtual int compare (const void*, const void*)
   {
      int first = *(int*)first_arg;
      int second = *(int*)second_arg;
      if ( first < second )
      {
         return -1;
      }
      else if ( first == second )
      {
         return 0;
      }
      else
      {
         return 1;
      }
   }
};
```

and then you could pass in a pointer to an instance of the AscendSorter to cpp_qsort to sort integers in ascending order.

**But Are You Really Not Using Function Pointers?**

Virtual functions are implemented behind the scenes using function pointers, so you really are using function pointers--it just happens that the compiler makes the work easier for you. Using

polymorphism can be an appropriate strategy (for instance, it's used by Java), but it does lead to the overhead of having to create an object rather than simply pass in a function pointer.

## Function Pointers Summary

### Syntax

Declare a function pointer as though you were declaring a function, except with a name like *foo instead of just foo:

```
void (*foo)(int);
```

### Initializing

You can get the address of a function simply by naming it:

```
void foo();
func_pointer = foo;
```

or by prefixing the name of the function with an ampersand:

```
void foo();
func_pointer = &foo;
```

### Invoking

Invoke the function pointed to just as if you were calling a function.

```
func_pointer( arg1, arg2 );
```

or you may optionally dereference the function pointer before calling the function it points to:

```
(*func_pointer)( arg1, arg2 );
```

### Benefits of Function Pointers

- Function pointers provide a way of passing around instructions for how to do something

- You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments
- This flexibility can also be achieved by using classes with virtual functions
- Classes

  In C++, a class is declared using the class keyword. The syntax of a class declaration is similar to that of a structure. Its general form is,class class-name

```
{
   private:
        // private functions and variables
  public:
       // public functions and variables
}
```

- object-list;In a class declaration the object-list is optional.The class-name is technically optional. From a practical point of view it is virtually always needed. The reason is that the class-name becomes a new type name that is used to declare objects of the class.Functions and variables declared inside the class declaration are said to be members of the class.

- By default, all member functions and variables are private to that class. This means that they are accessible by other members of that class.To declare public class members, the public keyword is used, followed by a colon. All functions and variables declared after the public specifier are accessible both by other members of the class and by any part of the program that contains the class.

- #include < iostream >

```
using namespace std;

// class declaration
class myclass
{
          // private members to myclass
      int a;
      public:
         // public members to myclass
     void set_a(int num);
     int get_a( );
```

```
};
```

- This class has one private variable, called a, and two public functions set_a( ) and get_a( ). Notice that the functions are declared within a class using their prototype forms. The functions that are declared to be part of a class are called member functions.

- Since a is private it is not accessible by any code outside myclass. However since set_a( ) and get_a( ) are member of myclass, they have access to a and as they are declared as public member of myclass, they can be called by any part of the program that contains myclass.

- The member functions need to be defined. You do this by preceding the function name with the class name followed by two colons (:: are called scope resolution operator). For example, outside the class declaration, you can declare the member function as

- // member functions declaration outside the class

```
void myclass::set_a(int num)
{
    a=num;
}
int myclass::get_a( )
{
   return a;
}
```

In general to declare a member function, you use this form:

```
return-type class-name::func-name(parameter- list)
{
        // body of function
}
```

Here the class-name is the name of the class to which the function belongs.

- The declaration of a class does not define any objects of the type myclass. It only defines the type of object that will be created when one is actually declared. To create an object, use the class name as type specifier. For example,

- // from previous examples

```
void main( )
{
        myclass ob1, ob2;    //these are object of type myclass
                // ... program code
}
```

- Remember that an object declaration creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not.

- Once an object of a class has been created, your program can reference its public members by using the dot operator in much the same way that structure members are accessed. Assuming the preceding object declaration, here some examples,

- ob1.set_a(10); // set ob1's version of a to 10

```
ob2.set_a(99); // set ob2's version of a to 99
cout << ob1.get_a( ); << "\n";
cout << ob2.get_a( ); << "\n";
ob1.a=20; // error cannot access private member
ob2.a=80; // by non-member functions.
...
```

There can be public variables, for example

```
#include < iostream >
using namespace std;
// class declaration
class myclass
{
 public:
      int a; //a is now public
            // and there is no need for set_a( ), get_a( )
};
int main( )
{
     myclass ob1, ob2;
     // here a is accessed directly
     ob1.a = 10;
```

```
    ob2.a = 99;
    cout << ob1.a << "\n";
    cout << ob1.a << "\n";
return 0;
}
```

It is important to remember that although all objects of a class share their functions, each object creates and maintains its own data.

- Constructors and Destructor
- Constructors
- When applied to real problems, virtually every object you create will require some sort of initialisation. C++ allows a constructor function to be included in a class declaration. A class's constructor is called each time an object of that class is created. Thus, any initialization to be performed on an object can be done automatically by the constructor function.
- A constructor function has the same name as the class of which it is a part a part and has not return type. Here is a short example,
- 

```
#include < iostream >
using namespace std;
// class declaration
class myclass
{
        int a;
  public:

        myclass( ); //constructor
        void show( );
};

myclass::myclass( )
{
cout << "In constructor\n";
a=10;
}
```

```
myclass::show( )
{
    cout << a;
}

int main( )
{
     int ob; // automatic call to constructor
    ob.show( );
    return 0;
}
```

In this simple example the constructor is called when the object is created, and the constructor initialises the private variable a to 10.For a global object, its constructor is called once, when the program first begins execution.For local objects, the constructor is called each time the declaration statement is executed.

- Destructors
- The complement of a constructor is the destructor. This function is called when an object is destroyed. For example, an object that allocates memory when it is created will want to free that memory when it is destroyed.
- The name of a destructor is the name of its class preceded by a ~. For example,
- ```
  myclass::~myclass( )
  {
     cout << "Destructing...\n";
  } // ...
  ```

  A class's destructor is called when an object is destroyed.Local objects are destroyed when they go out of scope. Global objects are destroyed when the program ends.It is not possible to take the address of either a constructor or a destructor.

- Note that having a constructor or a destructor perform actions not directly related to initialisation or orderly destruction of an object makes for very poor programming style and should be avoided.
  ```

- Constructors that take parameters

  It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments.

- ```
  #include < iostream >
  using namespace std;
  // class declaration
  class myclass
  {
          int a;
    public:
          myclass(int x); //constructor
          void show( );
  };


  myclass::myclass(int x)
  {
       cout << "In constructor\n";
       a=x;
  }


  void myclass::show( )
  {
       cout << a <<"\n";
  }
  int main( )
  {
       myclass ob(4);
       ob.show( );

  return 0;
  ```
  ```

}

Pay particular attention to how ob is declared in main( ). The value 4, specified in the parentheses following ob, is the argument that is passed to myclass( )'s parameter x that is used to initialise a. Actually, the syntax is shorthand for this longer form:

- myclass ob = myclass(4);

Unlike constructor functions, destructors cannot have parameters.Although the previous example has used a constant value, you can pass an object's constructor any valid expression, including variables.

## OBJECT

- So far, you have been accessing members of an object by using the dot operator. This is the correct method when you are working with an object. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed.You declare an object pointer just as you declare a pointer to any other type of variable. Specify its class name, and then precede the variable name with an asterisk.To obtain the address of an object, precede the object with the & operator, just as you do when taking the address of any other type of variable.
- Just as pointers to other types, an object pointer, when incremented, will point to the next object of its type. Here a simple example,
- #include < iostream >

```
using namespace std;
class myclass
{
        int a;
    public:
        myclass(int x); //constructor
        int get( );
};
myclass::myclass(int x)
{
    a=x;
}
int myclass::get( )
```

```
{
        return a;
}
int main( )
{
        myclass ob(120); //create object
        myclass *p; //create pointer to object
        p=&ob; //put address of ob into p
        cout << "value using object: " << ob.get( );
        cout << "\n";
        cout << "value using pointer: " << p->get( );
        return 0;

}
```

Notice how the declaration

```
myclass *p;
```

creates a pointer to an object of myclass. It is important to understand that creation of an object pointer does not create an object. It creates just a pointer to one. The address of ob is put into p by using the statement:

- 

```
p=&ob;
```

- Finally, the program shows how the members of an object can be accessed through a pointer.We will come back to object pointer later. For the moment, here are several special features that relate to them.

- Assigning object

- 

One object can be assigned to another provided that both are of the same type. By default, when one object is assigned to another, a bitwise copy of all the data members is made. For example, when an object called o1 is assigned to an object called o2 , the contents of all o1's data are copied into the equivalent members of o2.

```
//an example of object assignment.

//...
```

```
class myclass
{
    int a, b;
    public:
    void set(int i, int j) { a = i; b = j; };
    void show( ) { cout << a << " " << b << "\n"; }
};
int main( )
{

    myclass o1, o2;
    o1.set(10, 4);
    //assign o1 to o2
    o2 = o1;

    o1.show( );
    o2.show( );
    return 0;
}
```

Thus, when run this program displays

10 4

10 4

Remember that assignment between two objects simply makes the data, in those objects, identical. The two objects are still completely separate.

Only object of the same type can by assign. Further it is not sufficient that the types just be physically similar - their type names must be the same:

```
// This program has an error
class myclass
{
        int a, b;
  public:
        void set(int i, int j) { a = i; b = j; };
        void show( ) { cout << a << " " << b << "\n"; }
};

/* This class is similar to myclass but uses a different
type name and thus appears as a different type to
the compiler
*/
class yourclass
{
      int a, b;
   public:
      void set(int i, int j) { a = i; b = j; };
      void show( ) { cout << a << " " << b << "\n"; }
```

```
};
int main( )
 {
       myclass o1;
      yourclass o2;
      o1.set(10, 4);
      o2 = o1;                    //ERROR objects not of same type


      o1.show( );
     o2.show( );
return 0;
}
```

It is important to understand that all data members of one object are assigned to another when assignment is performed. This included compound data such as arrays. But be careful not to destroy any information that may be needed later.

Passing object to functions

Objects can be passed to functions as arguments in just the same way that other types of data are passed. Simply declare the function's parameter as a class type and then use an object of that class as an argument when calling the function. As with other types of data, by default all objects are passed by value to a function.

```
// ...
class samp
 {
          int i;
       public:
```

```
        samp(int n) { i = n; }

        int get_i( ) { return i; }

};

// Return square of o.i

int sqr_it(samp o)

{

    return o.get_i( )* o.get_i( );

}

int main( )

{

    samp a(10), b(2);

    cout << sqr_it(a) << "\n";

    cout << sqr_it(b) << "\n";

return 0;

}
```

As stated, the default method of parameter passing in C++, including objects, is by value. This means that a bitwise copy of the argument is made and it is this copy that is used by the function. Therefore, changes to the object inside the function do not affect the object in the call.

As with other types of variables the address of an object can be passed to a function so that the argument used in the call can be modify by the function.

```
// Set o.i to its square.
// This affect the calling argument

void sqr_it(samp *o)

{
```

```
    o->set(o->get_i( )*o->get_i( ));
}
// ...
int main( )
 {
    samp a(10);
    sqr_it(&a); // pass a's address to sqr_it
// ...
}
```

- Notice that when a copy of an object is created because it is used as an argument to a function, the constructor function is not called. However when the copy is destroyed (usually by going out of scope when the function returns), the destructor function is called.
- Be careful, the fact that the destructor for the object that is a copy of the argument is executed when the function terminates can be a source of problems. Particularly, if the object uses as argument allocates dynamic memory and frees that that memory when destroyed, its copy will free the same memory when its destructor is called.
- One way around this problem of a parameter's destructor function destroying data needed by the calling argument is to pass the address of the object and not the object itself. When an address is passed no new object is created and therefore no destructor is called when the function returns.
- A better solution is to use a special type of constructor called copy constructor, which we will see later on.
- Returning object from functions
- Functions can return objects. First, declare the function as returning a class type. Second, return an object of that type using the normal return statement.
- Remember that when an object is returned by a function, a temporary object is automatically created which holds the return value. It is this object that is actually returned by the function. After the value is returned, this object is destroyed. The destruction of the temporary object might cause unexpected side effects in some situations (e.g. when freeing dynamically allocated memory).

```
//Returning an object

// ...

class samp

 {


        char s[80];
    public:

        void show( ) { cout << s << "\n"; }

        void set(char *str) { strcpy(s, str); }

};


//Return an object of type samp

samp input( )

{

      char s[80];

      samp str;

      cout << "Enter a string: ";

      cin >> s;

      str.set(s);


return str;

}
```

```
int main( )
{
    samp ob;
        //assign returned object to ob
    ob = input( );
    ob.show( );

return 0;
}
```

## Friend functions:

- There will be time when you want a function to have access to the private members of a class without that function actually being a member of that class. Towards this, C++ supports friend functions. A friend function is not a member of a class but still has access to its private elements.
  Friend functions are useful with operator overloading and the creation of certain types of I/O functions. A friend function is defined as a regular, nonmember function. However, inside the class declaration for which it will be a friend, its prototype is also included, prefaced by the keyword friend. To understand how this works, here a short example:

```
//Example of a friend function
// ...
class myclass
{
        int n, d;
    public:
        myclass(int i, int j)
            {
              n = i;
```

```
        d = j;
    }

//declare a friend of myclass

friend int isfactor(myclass ob);
};


/* Here is friend function definition. It returns true
if d is a factor of n. Notice that the keyword friend
is not used in the definition of isfactor( ).
*/


int isfactor(myclass ob)
{
    if ( !(ob.n % ob.d) ) return 1;
    else return 0;
}


int main( )
{
    myclass ob1(10, 2), ob2(13, 3);
    if (isfactor(ob1))
        cout << "2 is a factor of 10\n";
    else
        cout << "2 is not a factor of 10\n";
    if (isfactor(ob2))
        cout << "3 is a factor of 13\n";
```

```
        else
                cout << "3 is not a factor of 13\n";
return 0;
}
```

- It is important to understand that a friend function is not a member of the class for which it is a friend. Thus, it is not possible to call a friend function by using an object name and a class member access operator (dot or arrow). For example, what follows is wrong.
-
- ob1.isfactor( ); //wrong isfactor is not a member function

- Instead friend functions are called just like regular functions.Because friends are not members of a class, they will typically be passed one or more objects of the class for which they are friends. This is the case with isfactor( ). It is passed an object of myclass, called ob. However, because isfactor( ) is a friend of myclass, it can access ob's private members. If isfactor( ) had not been made a friend of myclass it would not have access to ob.d or ob.n since n and d are private members of myclass.
- A friend function is not inherited. That is, when a base class includes a friend function, that friend function is not a friend function of the derived class.
- A friend function can be friends with more than one class. For example,
- // ...

```
class truck; //This is a forward declaration
class car
{
        int passengers;
        int speed;
  public:
    car(int p, int s) { passengers = p; speed =s; }
    friend int sp_greater(car c, truck t);
};
```

```
class truck
{
        int weight;
        int speed;
   public:
        truck(int w, int s) { weight = w; speed = s; }
        friend int sp_greater(car c, truck t);
};



int sp_greater(car c, truck t)
{
     return c.speed - t.speed;
}
int main( )
{
// ...
}
```

This program also illustrates one important element: the forward declaration (also called a forward reference), to tell the compiler that an identifier is the name of a class without actually declaring it. A function can be a member of one class and a friend of another class. For example,

```
// ...
class truck; // forward declaration
class car
 {
      int passengers;
      int speed;
  public:
       car(int p, int s) { passengers = p; speed =s; }
       int sp_greater( truck t);
};
class truck
```

```
{
    int weight;
    int speed;

 public:
    truck(int w, int s) { weight = w; speed = s; }
    //note new use of the scope resolution operator
    friend int car::sp_greater( truck t);
};

int car::sp_greater(truck t)
{
   return speed - t.speed;
}
int main( )
{
  // ...
}
```

One easy way to remember how to use the scope resolution operation it is never wrong to fully specify its name as above in class truck,

```
friend int car::sp_greater( truck t);
```

However, when an object is used to call a member function or access a member variable, the full name is redundant and seldom used. For example,

```
// ...
int main( ) {
int t;
car c1(6, 55);
truck t1(10000, 55);
t = c1.sp_greater(t1); //can be written using the
//redundant scope as
t = c1.car::sp_greater(t1);
//...
```

```
}
```

However, since c1 is an object of type car the compiler already knows that sp_greater( ) is a member of the car class, making the full class specification unnecessary.

## ARRAYS, POINTERS, AND REFERENCES

Objects are variables and have the same capabilities and attributes as any other type of variables. Therefore, it is perfectly acceptable for objects to be arrayed. The syntax for declaring an array of objects is exactly as that used to declare an array of any other type of variable. Further, arrays of objects are accessed just like arrays of other types of variables.

```
#include < iostream >
using namespace std;
class samp
 {
          int a;
      public:
          void set_a(int n) {a = n;}
          int get_a( ) { return a; }
};


int main( )
{
        samp ob[4];          //array of 4 objects
        int i;

      for (i=0; i<4; i++) ob[i].set_a(i);
      for (i=0; i<4; i++) cout << ob[i].get_a( ) << " ";
      cout << "\n";
return 0;
}
```

If the class type include a constructor, an array of objects can be initialised,

```
// Initialise an array
#include < iostream >
using namespace std;
class samp
{
      int a;
      public:
      samp(int n) {a = n; }
       int get_a( ) { return a; }
};


int main( )
{
      samp ob[4] = {-1, -2, -3, -4};
      int i;
      for (i=0; i<4; i++) cout << ob[i].get_a( ) << " ";
      cout << "\n"
return 0;
}
```

You can also have multidimensional arrays of objects. Here an example

```
// Create a two-dimensional array of objects
// ...
class samp
{
       int a;
       public:
       samp(int n) {a = n; }
       int get_a( ) { return a; }
```

```
};

int main( )
{
     samp ob[4][2] = {
                              1, 2,
                              3, 4,
                              5, 6,
                              7, 8 };
    int i;

    for (i=0; i<4; i++)
    {
        cout << ob[i][0].get_a( ) << " ";
        cout << ob[i][1].get_a( ) << "\n";
     }
    cout << "\n";
return 0;
}
```

This program displays,

1 2

3 4

5 6

7 8

When a constructor uses more than one argument, you must use the alternative format,

```
// ...
class samp
{
          int a, b;
     public:
        samp(int n, int m) {a = n; b = m; }
```

```
    int get_a( ) { return a; }
    int get_b( ) { return b; }
};

int main( )
{
    samp ob[4][2] = {
                        samp(1, 2), samp(3, 4),
                        samp(5, 6), samp(7, 8),
                        samp(9, 10), samp(11, 12),
                        samp(13, 14), samp(15, 16)
                       };
// ...
```

Note you can always the long form of initialisation even if the object takes only one argument. It is just that the short form is more convenient in this case.

Using pointers to objects

As you know, when a pointer is used, the object's members are referenced using the arrow (- >) operator instead of the dot (.) operator.

Pointer arithmetic using an object pointer is the same as it is for any other data type: it is performed relative to the type of the object. For example, when an object pointer is incremented, it points to the next object. When an object pointer is decremented, it points to the previous object.

```
// Pointer to objects
// ...
class samp {
int a, b;
public:
samp(int n, int m) {a = n; b = m; }
int get_a( ) { return a; }
int get_b( ) { return b; }
};
int main( ) {
samp ob[4] = {
samp(1, 2),
samp(3, 4),
samp(5, 6),
```

```
samp(7, 8)
};
int i;
samp *p;
p = ob; // get starting address of array
for (i=0; i<4; i++) {
cout << p->get_a( ) << " ";
cout << p->get_b( ) << "\n";
p++; // advance to next object
}
// ...
```



# Reference and Bibliography


1. Object Oriented Programming with C++ E.Balaguruswamy TMH 6th Edition, 2013
2. ObjectOriented Programming with C++ Robert Lafore Galgotia publication 2010
3. ObjectOriented Programming with C++ Sourav Sahay Oxford University 2006
4. Preece, J. Rogers, Y. and Sharp,H., 2007. Interaction Design: Beyond Human Computer Interaction. 2nd ed. New York: John Wiley & Sons, Inc.

5. Preece, J. Rogers, Y. and Sharp,H., 2001. Interaction Design: Beyond Human Computer Interaction. New York: John Wiley & Sons, Inc.

6. Andrew, G and Drew, P, 2009, Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture, Journal of Database Management, 20(1), 1- 24, January-March 2009.