

8. Functions

Function is a block of statements that performs some operations. All C++ programs have at least one function – function called "main()". This function is entry-point of your program. But you can define a lot of new functions, which will be used in your programs.

8.1 Advantages of using functions

1. You can divide your program in logical blocks. It will make your code clear and easy to understand.
2. Use of function avoids typing same pieces of code multiple times. You can call a function to execute same lines of code multiple times without re-writing it.
3. Individual functions can be easily tested.
4. In case of any modification in the code you can modify only the function without changing the structure of the program.

8.2 Structure of a function

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.

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

8.3 Declaring and defining functions

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.

- J 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.
- J 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)
{
    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 important 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.

8.4 Return Statement

Function can return single value to the calling program using return statement. In the above program, the value of **add** is returned from user-defined function to the calling program using statement below:

```
return add;
```

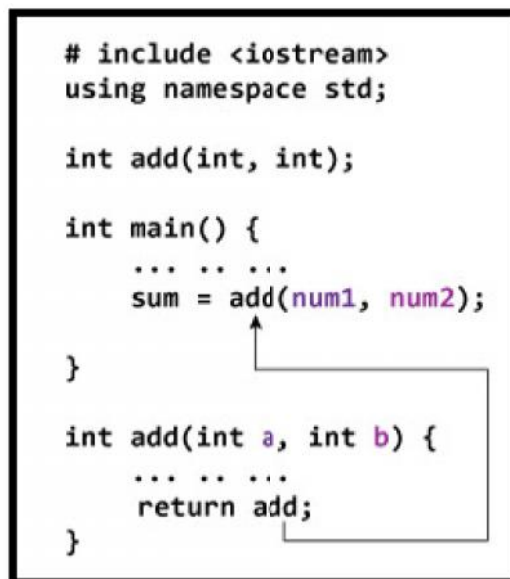
The figure below demonstrates the working of return statement.

In the above program, the value of **add** inside user-defined function is returned to the calling function. The value is then stored to a **variable** **sum**. Notice that, the variable returned, that is, **add** is of type **int** and also **sum** is of **int** type. Also notice that, the return type of a function is defined in function declarator **int add(int a, int b)**. The **int** before **add(int a, int b)** means that function should return a value of type **int**. If no value is returned to the calling function then, **void** should be used.

Passing Arguments

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.

In above example, two variables, **num1** and **num2** are passed to function during function call. These arguments are known as actual arguments. The value of **num1** and **num2** are initialized to variables **a** and **b** respectively. These arguments **a** and **b** are called formal arguments. This is demonstrated in figure below:

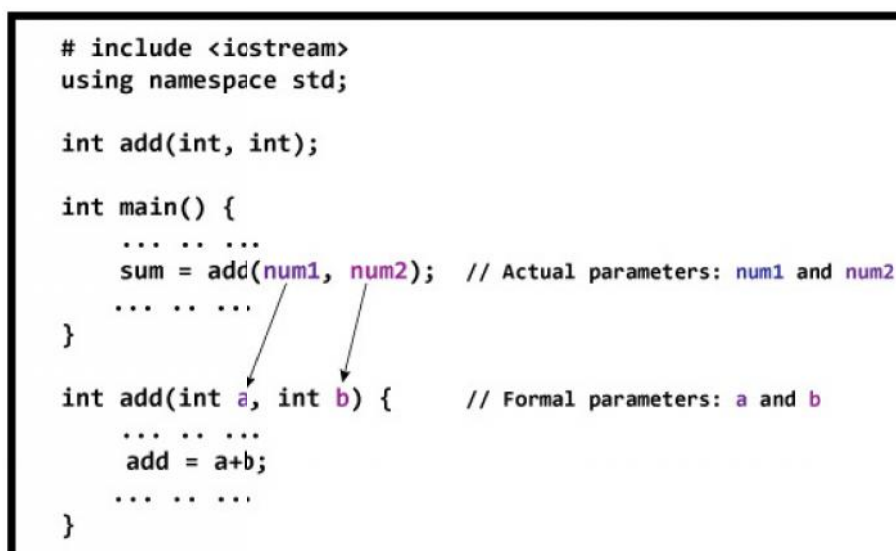


```
# include <iostream>
using namespace std;

int add(int, int);

int main() {
    ... ..
    sum = add(num1, num2);
}

int add(int a, int b) {
    ... ..
    return add;
}
```



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

int add(int, int);

int main() {
    ... ..
    sum = add(num1, num2); // Actual parameters: num1 and num2
    ... ..
}

int add(int a, int b) { // Formal parameters: a and b
    ... ..
    add = a+b;
    ... ..
}
```

Notes on passing arguments

-) The numbers of actual arguments and formal argument should be same. (Exception: Function Overloading)
-) The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
-) You may call function without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
-) In above program, both arguments are of **int** type. But it's not necessary to have both arguments of same type.

8.5 Formal and actual arguments

The arguments may be classified under two groups, actual and formal arguments.

(a) Actual arguments An actual argument is a variable or an expression contained in a function call that replaces the formal parameter which is a part of the function declaration.

Sometimes, a function may be called by a portion of a program with some parameters and these parameters are known as the actual arguments.

For example,

```
#include <iostream.h>
void main ()
{
    int x,y;
    void output (int x, int y) ; // function declaration
    _____
    _____

    output (x,y) ; // x and y are the actual arguments
}
```

(b) Formal arguments Formal arguments are the parameters presents in a function definition which may also be called as dummy arguments or the parametric variables. When the function is invoked, the formal parameters are replaced by the actual parameters.

For example,

```
#include <iostream.h>
void main ()
{
    int x,y;
    void output (int x, int y) ;

    output (x,y) ;
}
void output (int a, int b) // formal or dummy arguments
{
    //body of the function
}
```

Formal arguments may be declared by the same name or by different names in calling a portion of the program or in a called function but the data types should be the same in both blocks.

For example, the following function declaration is invalid.

```
#include <iostream.h>
void main ()
{
    void funct (int x, int y, char s1, char s2) ;
    int x,y;
    char s1, s2;

    _____
    _____

    funct (x,y,s1,s2) ;
}
funct (char c1, char c2, int a, int b) // data type mismatch
{
    // body of a function
}
```

8.6 const argument

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while declared.

```
int main
{
    const int i = 10;
    const int j = i+10; // Works fine
    i++; // This leads to Compile time error
}
```

In this program we have made `i` as constant, hence if we try to change its value, compile time error is given. Though we can use it for substitution.

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)
{
    i++; // Error
}
const int g()
{
    return 1;
}
```

8.7 Default arguments

When you define a function, you 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.

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.h>
#include <conio.h>

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

    result = a + b;

    return (result);
}

int main ()
{
    int a = 100;
    int b = 200;
    int result;

    result = sum(a, b);
    cout << "Total value is :" << result << endl;
    result = sum(a);
    cout << "Total value is :" << result << endl;

    return 0;
}
```

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

```
Total value is :300
Total value is :120
```

8.8 Concept of reference variable

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example:

```
int i = 17;
```

We can declare reference variables for i as follows.

```
int& r = i;
```

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double:

```

#include <iostream.h>
#include<conio.h>

int main ()
{
    // declare simple variables
    int i;
    double d;

    // declare reference variables
    int& r = i;
    double& s = d;

    i = 5;
    cout << "Value of i : " << i << endl;
    cout << "Value of i reference : " << r << endl;

    d = 11.7;
    cout << "Value of d : " << d << endl;
    cout << "Value of d reference : " << s << endl;

    return 0;
}

```

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

```

Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7

```

8.9 Calling / Invoking 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.h>
#include<conio.h>

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

int main ()
{

```

```

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;
}

int max(int num1, int num2)
{
    int result;

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

    return result;
}

```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the 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
<u>Call by value</u>	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.
<u>Call by pointer</u>	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.
<u>Call by reference</u>	This method copies the reference of an argument into the formal parameter. Inside the function, the reference 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.

8.10 Call by value

The **call by value** method of passing arguments to a function 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.

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. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:

```
#include <iostream.h>
#include <conio.h>

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

void main ()
{
    int a = 100, 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.
    swap(a, b);

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

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

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

Which shows that there is no change in the values though they had been changed inside the function.

8.11 Call by reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference 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 reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference 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;
    x = y;
    y = temp;

    return;
}
```

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

```
#include <iostream.h>
#include <conio.h>

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

int main ()
{
    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 using variable reference.*/
    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 the 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
```

8.12 Library functions

Library functions are built in function that are defined in the C++ library. Function prototype is present in header files so we need to include specific header files to use library functions. These functions can be used by simply calling the function. Some library functions are `pow()`, `sqrt()`, `strcpy()`, `toupper()`, `isdigit()`, etc.

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the `#include` directive at the top of the program file, e.g. for common mathematical calculations we include the file `cmath` with the `#include <cmath>` directive which contains the *function prototypes* for the mathematical functions in the `cmath` library

Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

Function	Description
<code>sqrt(x)</code>	square root
<code>sin(x)</code>	trigonometric sine of x (in radians)
<code>cos(x)</code>	trigonometric cosine of x (in radians)
<code>tan(x)</code>	trigonometric tangent of x (in radians)
<code>exp(x)</code>	exponential function
<code>log(x)</code>	natural logarithm of x (base e)
<code>log10(x)</code>	logarithm of x to base 10
<code>fabs(x)</code>	absolute value (unsigned)
<code>ceil(x)</code>	rounds x up to nearest integer
<code>floor(x)</code>	rounds x down to nearest integer
<code>pow(x,y)</code>	x raised to power y

Random numbers

Other **header files** which contain the function prototypes of commonly used functions include `cstdlib` and `time`. These contain functions for generating random numbers and for manipulating time and dates respectively.

The function `random()` *randomly* generates an integer between 0 and the maximum value which can be stored as an integer. Every time the function is called:

```
randomNumber = random();
```

8.13 Recursion

In many programming languages including C++, it is possible to call a function from a same function. This function is known as recursive function and this programming technique is known as recursion.

To understand recursion, you should have knowledge of two important aspects:

- 1) In recursion, a function calls itself but you shouldn't assume these two functions are same function. They are different functions although they have same name.
- 2) **Local variables**: Local variables are variables defined inside a function and has scope only inside that function. In recursion, a function call itself but these two functions are different functions (You can imagine these functions are function1 and function 2. The local variables inside function1 and function2 are also different and can only be accessed within that function.

Consider this example to find factorial of a number using recursion:

```
#include <iostream.h>
#include <conio.h>

int factorial(int);

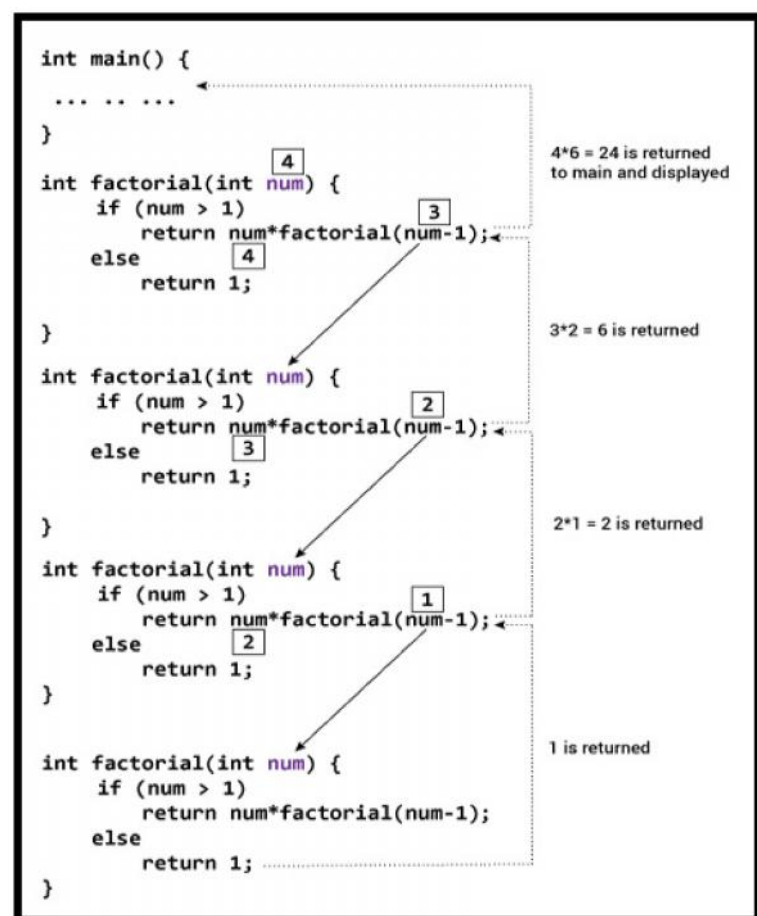
int main() {
    int n;
    cout<<"Enter a number to find factorial: ";
    cin>>n;
    cout<<"Factorial of "<<n<<" = "<<factorial(n);
    return 0;
}

int factorial(int n) {
    if (n>1) {
        return n*factorial(n-1);
    }
    else {
        return 1;
    }
}
```

Output

```
Enter a number to find factorial: 4
Factorial of 4 = 24
```

Explanation: How recursion works?



Suppose user enters 4 which is passed to function **factorial()**. Here are the steps involved:

-) In first **factorial()** function, test expression inside **if statement** is true. The statement **return num*factorial(num-1);** is executed, which calls second **factorial()** function and argument passed is **num-1** which is 3.
-) In second **factorial()** function, test expression inside **if statement** is true. The statement **return num*factorial(num-1);** is executed, which calls third **factorial()** function and argument passed is **num-1** which is 2.
-) In third **factorial()** function, test expression inside **if statement** is true. The statement **return num*factorial(num-1);** is executed, which calls fourth **factorial()** function and argument passed is **num-1** which is 1.
-) The fourth **factorial()** function, test expression inside **if statement** is false. The statement **return 1;** is executed, which returns 1 to third **factorial()** function.

-) The third **factorial()** function returns 2 to second **factorial()** function.
-) The second **factorial()** function returns 6 to first **factorial()** function.
-) Finally, first **factorial()** function returns 24 to the **main()** function and is displayed.

What is recursion? The simple answer is, it's when a function calls itself. But how does this happen? Why would this happen, and what are its uses?

When we talk about recursion, we are really talking about creating a loop. Let's start by looking at a basic loop.

```
1 for(int i=0; i<10; i++) {
2     cout << "The number is: " << i << endl;
3 }
```

For those who don't yet know, this basic loop displays the sentence, "The number is: " followed by the value of 'i'. Like this.

```
The number is: 0
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
```

Inside the 'for loop' declaration we have the integer variable 'i' and have its starting value of 0. So the first time the sentence is displayed it reads, "The number is: 0". The part of the 'for loop' declaration that is 'i++' tells the program that each time the loop repeats, the value of 'i' should be increased by 1. So, the next time the sentence is displayed it reads, "The number is: 1". This cycle will continue to repeat for as long as the value of 'i' is less than 10. The last sentence displayed would read, "The number is: 9". As you can see the basic 'for loop' has three parts to its declaration, a starting value, what must remain true in order to continue repeating, and a modifying expression. Everything that is contained within the {braces} is what the program performs. Cout stands for console out, and prints words or characters to the screen. So what does this have to do with recursion? Remember recursion is a loop. What if I did not want to just print a message to the screen? A loop can be used to perform other tasks as well.

In the following code is the same loop as above only now it is being used to call a function.

```
1 #include <iostream.h>
2 #include<conio.h>
3
4 void numberFunction(int i) {
5     cout << "The number is: " << i << endl;
6 }
7
8 int main() {
9     for(int i=0; i<10; i++) {
10         numberFunction(i);
11     }
12     return 0;
13 }
```

I have declared a void function, which means it returns nothing, and takes a parameter of 'int i'. The function is named 'numberFunction' and as you can see, all it does is display the sentence, "The number is: " followed by the current value of 'i'. The function is called into use by the 'for loop', which continually calls the function as long as the value of 'i' is less than 10.

Now with recursion, we won't need to use a 'for loop' because we will set it up so that our function calls itself. Let's recreate this same program one more time, only this time we will do it without a 'for loop'. We will use a recursion loop instead, like this.

```

1 #include <iostream.h>
2 #include <conio.h>
3
4 void numberFunction(int i) {
5     cout << "The number is: " << i << endl;
6     i++;
7     if(i<10) {
8         numberFunction(i);
9     }
10 }
11
12 int main() {
13
14     int i = 0;
15     numberFunction(i);
16
17     return 0;
18 }

```

We did it! We used recursion! You can see the call to 'numberFunction' is made only once in the main part of the program but it keeps getting called again and again from within the function itself, for as long as 'i' is less than 10

8.14 Storage classes

Storage class of a variable defines the lifetime and visibility of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible. There are five types of storage classes in C++. They are:

1. Automatic
2. External
3. Static
4. Register
5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
    int mount;
    auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
    register int miles;
}
```

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream.h>

void func(void);
static int count = 10;

main()
{
    while(count-->0)
    {
        func();
    }
    return 0;
}

// Function definition
void func( void )
{
    static int i = 5;
    i++;
    std::cout << "i is " << i ;
}
```

```
std::cout << " and count is " << count << std::endl;
}
```

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

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream.h>

int count ;
extern void write_extern();

main()
{
    count = 5;
    write_extern();
}
```

Second File: support.cpp

```
#include <iostream.h>

extern int count;

void write_extern(void)
{
    std::cout << "Count is " << count << std::endl;
}
```