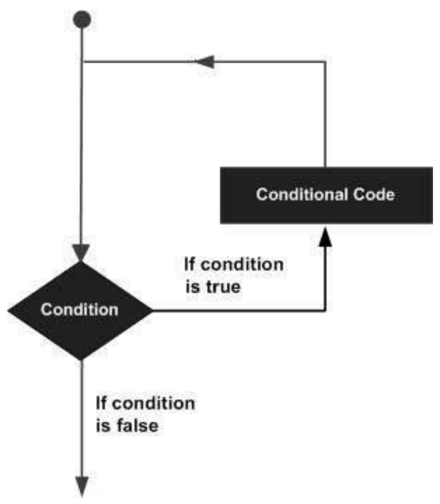
C++ Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

Sr.No	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop Like a ‘while’ statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Sr.No	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

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

int main () {
   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the 'for (;;)' construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

C++ while loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

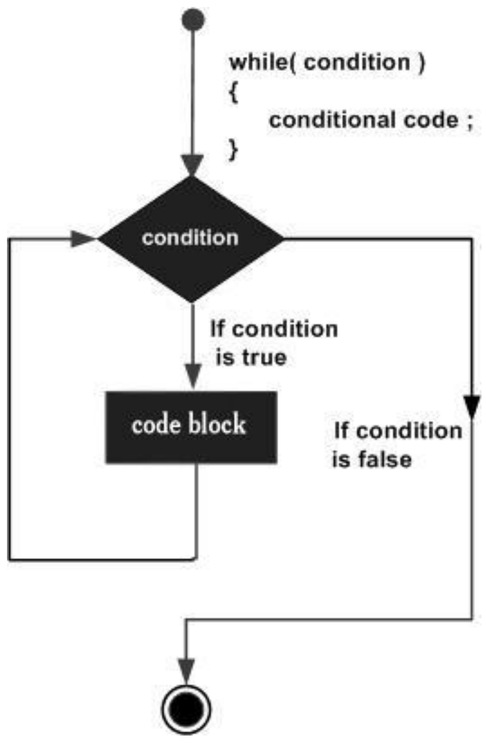
The syntax of a while loop in C++ is −

```
while(condition) {
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

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

int main () {
   // Local variable declaration:
   int a = 10;

   // while loop execution
   while( a < 20 ) {
      cout << "value of a: " << a << endl;
      a++;
   }

   return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C++ for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

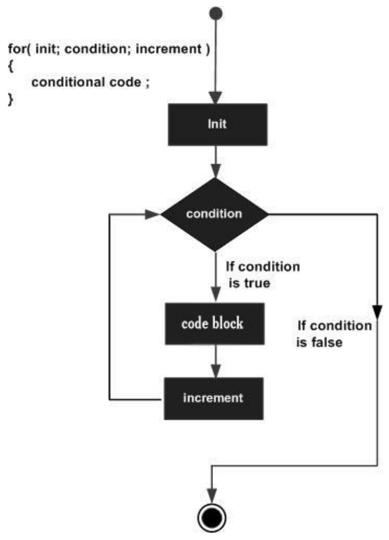
The syntax of a for loop in C++ is −

```
for ( init; condition; increment ) {
   statement(s);
}
```

Here is the flow of control in a for loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example

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

int main () {
   // for loop execution
   for( int a = 10; a < 20; a = a + 1 ) {
      cout << "value of a: " << a << endl;
   }

   return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C++ do...while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

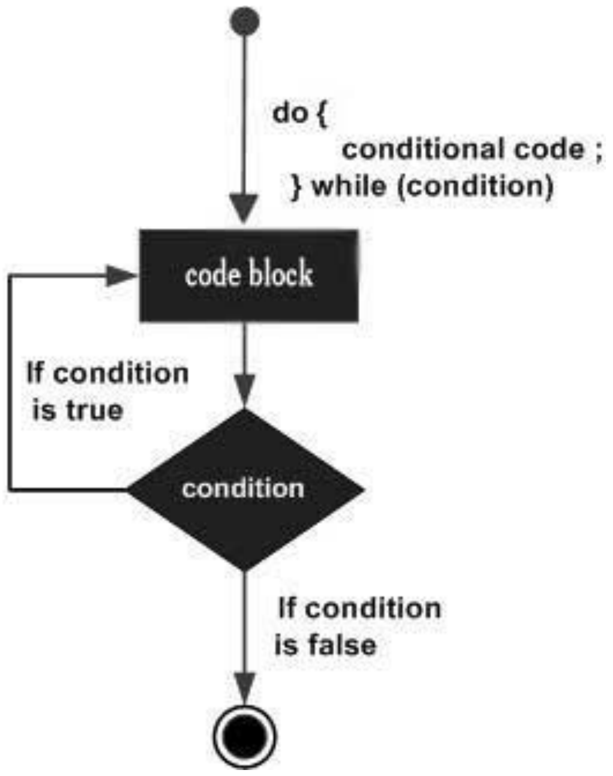
The syntax of a do...while loop in C++ is −

```
do {
   statement(s);
}
while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram



Example

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

int main () {
   // Local variable declaration:
   int a = 10;

   // do loop execution
   do {
      cout << "value of a: " << a << endl;
      a = a + 1;
   } while( a < 20 );

   return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

C++ Functions

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 such that 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 known 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 return the biggest of both −

```
// 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 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.

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 it's 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 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 −

Sr.No	Call Type & Description
1	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.
2	Call by Pointer 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.
3	Call by Reference 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.

Default Values for Parameters

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>
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 the following result −

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

C++ function 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; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

Now, let us call the function **swap()** by passing actual values 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.

   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 :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.

C++ function call by pointer

The **call by pointer** 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 pointer, 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 pointer 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 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
```

C++ function 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; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

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 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 −

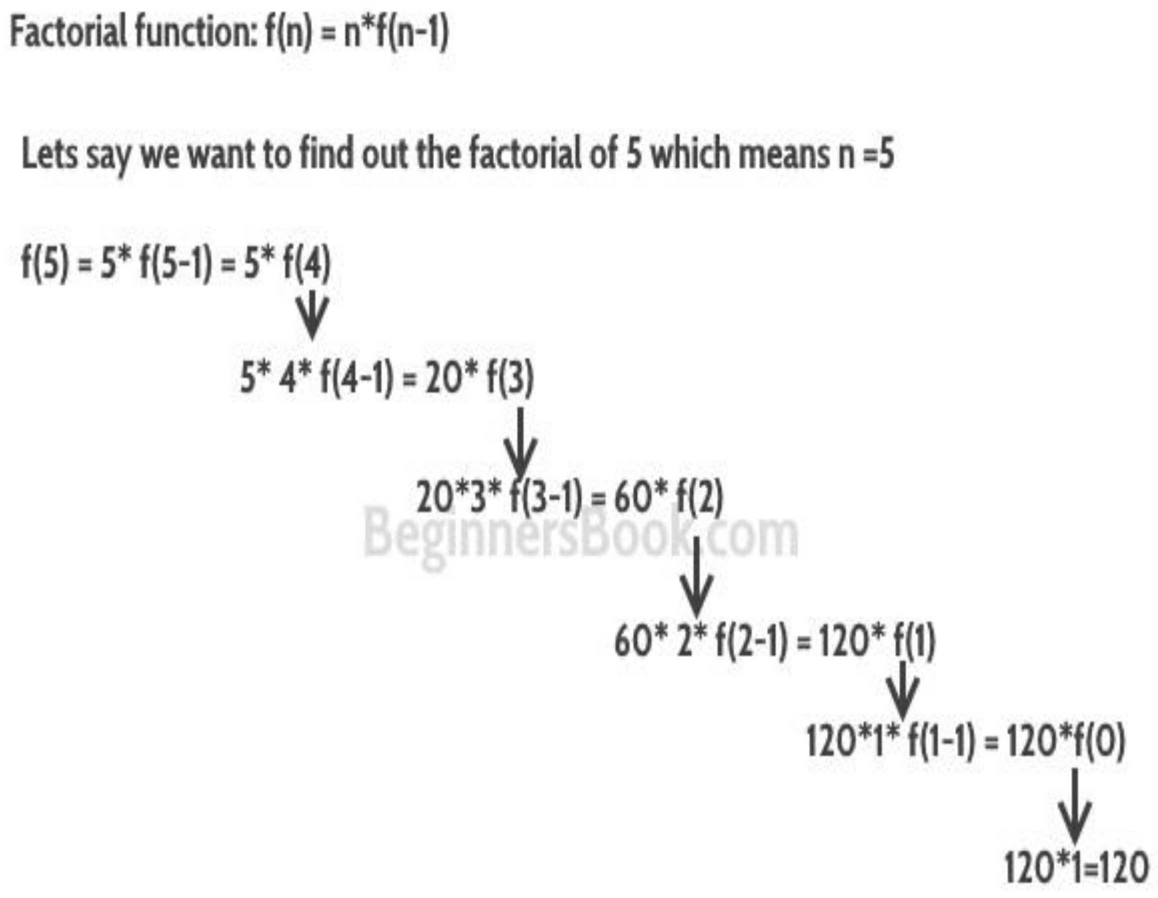
```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

C++ Recursion

The process in which a function calls itself is known as recursion and the corresponding function is called the **recursive function**. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.


Factorial function: f(n) = n*f(n-1)
Lets say we want to find out the factorial of 5 which means n =5
f(5) = 5* f(5-1) = 5* f(4)
5* 4* f(4-1) = 20* f(3)
20*3* f(3-1) = 60* f(2)
60* 2* f(2-1) = 120* f(1)
120*1* f(1-1) = 120*f(0)
120*1=120
BeginnersBook.com

Lets solve the problem using C++ program.

C++ recursion example: Factorial

```
#include <iostream>
using namespace std;
//Factorial function
int fact(int n){
   /* This is called the base condition, it is
    * very important to specify the base condition
    * in recursion, otherwise your program will throw
    * stack overflow error.
    */
   if (n <= 1)
        return 1;
   else
       return n*fact(n-1);
}
int main(){
   int num;
   cout<<"Enter a number: ";
   cin>>num;
   cout<<"Factorial of entered number: "<<fact(num);
   return 0;
}
```

Output:

```
Enter a number: 5
Factorial of entered number: 120
```

Base condition

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if (n <= 1)
        return 1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

Direct recursion vs indirect recursion

Direct recursion: When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

Indirect recursion: When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

Indirect Recursion Example in C++

```
#include <iostream>
using namespace std;
int fa(int);
int fb(int);
int fa(int n){
   if(n<=1)
      return 1;
   else
      return n*fb(n-1);
}
int fb(int n){
   if(n<=1)
      return 1;
   else
      return n*fa(n-1);
}
int main(){
   int num=5;
   cout<<fa(num);
   return 0;
}
```

Output:

```
120
```

Numbers in C++

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

Defining Numbers in C++

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++ −

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

int main () {
   // number definition:
   short  s;
   int    i;
   long   l;
   float  f;
   double d;

   // number assignments;
   s = 10;
   i = 1000;
   l = 1000000;
   f = 230.47;
   d = 30949.374;

   // number printing;
   cout << "short  s :" << s << endl;
   cout << "int    i :" << i << endl;
   cout << "long   l :" << l << endl;
   cout << "float  f :" << f << endl;
   cout << "double d :" << d << endl;

   return 0;
}
```

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

```
short  s :10
int    i :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

Math Operations in C++

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called **built-in** functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file **<cmath>**.

Sr.No	Function & Purpose
1	**double cos(double);** This function takes an angle (as a double) and returns the cosine.
2	**double sin(double);** This function takes an angle (as a double) and returns the sine.
3	**double tan(double);** This function takes an angle (as a double) and returns the tangent.
4	**double log(double);** This function takes a number and returns the natural log of that number.
5	**double pow(double, double);** The first is a number you wish to raise and the second is the power you wish to raise it t
6	**double hypot(double, double);** If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse.
7	**double sqrt(double);** You pass this function a number and it gives you the square root.
8	**int abs(int);** This function returns the absolute value of an integer that is passed to it.
9	**double fabs(double);** This function returns the absolute value of any decimal number passed to it.
10	**double floor(double);** Finds the integer which is less than or equal to the argument passed to it.

Following is a simple example to show few of the mathematical operations −

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

int main () {
   // number definition:
   short  s = 10;
   int    i = -1000;
   long   l = 100000;
   float  f = 230.47;
   double d = 200.374;

   // mathematical operations;
   cout << "sin(d) :" << sin(d) << endl;
   cout << "abs(i)  :" << abs(i) << endl;
   cout << "floor(d) :" << floor(d) << endl;
   cout << "sqrt(f) :" << sqrt(f) << endl;
   cout << "pow( d, 2) :" << pow(d, 2) << endl;

   return 0;
}
```

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

```
sign(d)      :-0.634939
abs(i)       :1000
floor(d)     :200
sqrt(f)      :15.1812
pow( d, 2 ) :40149.7
```

Random Numbers in C++

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is **rand()**, this function will only return a pseudo random number. The way to fix this is to first call the **srand()** function.

Following is a simple example to generate few random numbers. This example makes use of **time()** function to get the number of seconds on your system time, to randomly seed the rand() function −

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;
```

```
int main () {
   int i,j;

   // set the seed
   srand( (unsigned)time( NULL ) );

   /* generate 10  random numbers. */
   for( i = 0; i < 10; i++ ) {
      // generate actual random number
      j = rand();
      cout <<" Random Number : " << j << endl;
   }

   return 0;
}
```

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

```
Random Number : 1748144778
Random Number : 630873888
Random Number : 2134540646
Random Number : 219404170
Random Number : 902129458
Random Number : 920445370
Random Number : 1319072661
Random Number : 257938873
Random Number : 1256201101
Random Number : 580322989
```

C++ Arrays

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

```
double balance[10];
```

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

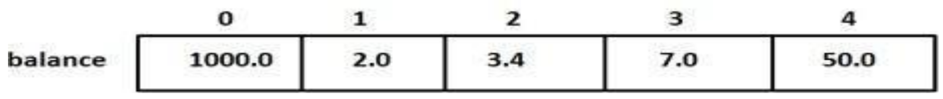
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

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

#include <iomanip>
using std::setw;

int main () {

   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // set element at location i to i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
   for ( int j = 0; j < 10; j++ ) {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result −

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer −

Sr.No	Concept & Description
1	Multi-dimensional arrays C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
3	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
4	Return array from functions C++ allows a function to return an array.

C++ Multi-dimensional Arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array

```
int threedim[5][10][4];
```

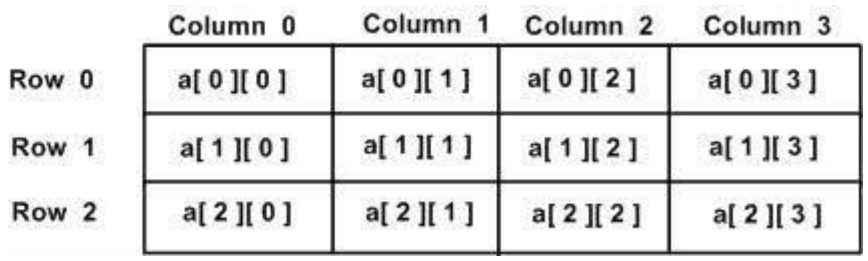
Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

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

int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

   // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ ) {

         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }

   return 0;
}
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

C++ Pointer to an Array

It is most likely that you would not understand this chapter until you go through the chapter related C++ Pointers.

So assuming you have bit understanding on pointers in C++, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration −

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** the address of the first element of **balance** −

```
double *p;
double balance[10];
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on. Below is the example to show all the concepts discussed above −

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

int main () {
   // an array with 5 elements.
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;

   p = balance;

   // output each array element's value
   cout << "Array values using pointer " << endl;

   for ( int i = 0; i < 5; i++ ) {
      cout << "*(p + " << i << ") : ";
      cout << *(p + i) << endl;
   }
   cout << "Array values using balance as address " << endl;

   for ( int i = 0; i < 5; i++ ) {
      cout << "*(balance + " << i << ") : ";
      cout << *(balance + i) << endl;
   }

   return 0;
}
```

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

```
Array values using pointer
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
Array values using balance as address
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```

In the above example, p is a pointer to double which means it can store address of a variable of double type. Once we have address in p, then ***p** will give us value available at the address stored in p, as we have shown in the above example.

C++ Passing Arrays to Functions

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

Formal parameters as a pointer as follows −

```
void myFunction(int *param) {
   .
   .
   .
}
```

Way-2

Formal parameters as a sized array as follows −

```
void myFunction(int param[10]) {
   .
   .
   .
}
```

Way-3

Formal parameters as an unsized array as follows −

```
void myFunction(int param[]) {
   .
   .
   .
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows −

```
double getAverage(int arr[], int size) {
  int i, sum = 0;
  double avg;
   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }
   avg = double(sum) / size;
   return avg;
}
```

Now, let us call the above function as follows −

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

// function declaration:
double getAverage(int arr[], int size);

int main () {
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

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

```
Average value is: 214.4
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

Passing multidimensional array to function

In this example we are passing a multidimensional array to the function square which displays the square of each element.

```
#include <iostream>
#include <cmath>
using namespace std;
/* This method prints the square of each
 * of the elements of multidimensional array
 */
void square(int arr[2][3]){
   int temp;
   for(int i=0; i<2; i++){
      for(int j=0; j<3; j++){
        temp = arr[i][j];
        cout<<pow(temp, 2)<<endl;
      }
   }
}
```

```
int main(){
   int arr[2][3] = {
       {1, 2, 3},
       {4, 5, 6}
   };
   square(arr);
   return 0;
}
```

```
1
4
9
16
25
36
```

Return Array from Functions in C++

C++ does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example −

```
int * myFunction() {
    .
    .
    .
}
```

Second point to remember is that C++ does not advocate to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.

Now, consider the following function, which will enter 5 numbers and return them using an array and call this function as follows −

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

// function to enter 5 numbers
int * print(){
       static int a[5]={1,2,3,4,5};

       return a;
}

// main function to call above defined function.
int main()
{
       // a pointer to an int.
       int * k;

       k = print();

       for(int j=0; j<5; j++)
       cout<< *(k+j)<<"\t";

       return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows −

```
1      2      3      4      5
```

C++ Strings

C++ provides following two types of string representations −

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

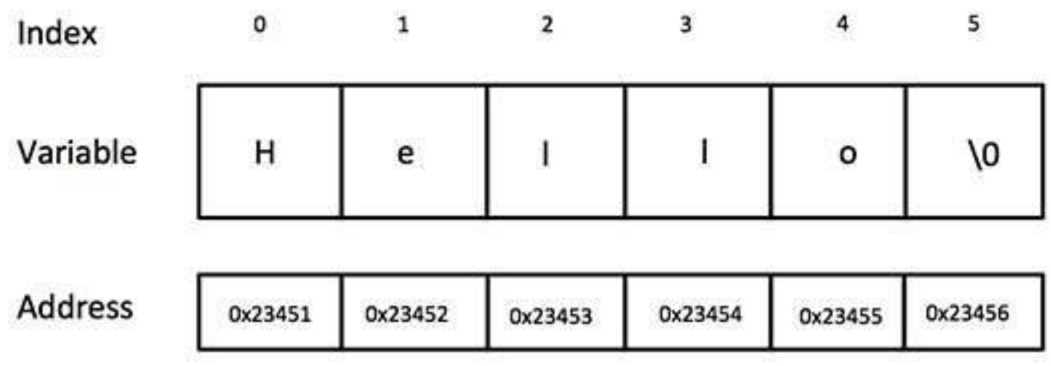
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

```
#include <iostream>
using namespace std;
int main () {
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   cout << "Greeting message: ";
   cout << greeting << endl;
   return 0;
}
```

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

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings −

Sr.No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions −

```
#include <iostream>
#include <cstring>
using namespace std;
int main () {
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;
   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;
   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;
   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

```
#include <iostream>
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

C++ Files and Streams

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.

3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();
   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```