About the content

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features. **C++** runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Some of the features & key-points to note about the programming language are as follows:

- **Simple**: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- **Machine Independent but Platform Dependent**: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.

- **Mid-level language**: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support**: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution**: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access**: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- **Object-Oriented**: One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.
- **Compiled Language**: C++ is a compiled language, contributing to its speed.

Applications of C++ Programming

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like softwares for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great softwares. I highly recommend you to learn C++ and contribute great softwares to the community.

Object-Oriented Programming

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

- Encapsulation
- Data hiding

- Inheritance
- Polymorphism

Standard Libraries

Standard C++ consists of three important parts −

- The core language giving all the building blocks including variables, data types and literals, etc.
- The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
- The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

Some interesting facts about C++:

Here are some awesome facts about C++ that may interest you:

1. The name of C++ signifies the evolutionary nature of the changes from C. "++" is the C increment operator.
2. C++ is one of the predominant languages for the development of all kind of technical and commercial software.
3. C++ introduces Object-Oriented Programming, not present in C. Like other things, C++ supports the four primary features of OOP: encapsulation, polymorphism, abstraction, and inheritance.
4. C++ got the OOP features from Simula67 Programming language.
5. A function is a minimum requirement for a C++ program to run.(at least main() function)

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods(functions). Let us now briefly look into what a class, object, methods, and instant variables mean.

- **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class** − A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
- **Methods(functions)** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Writing First C++ Program – Hello World Example

The "Hello World" program is the first step towards learning any programming language and is also one of the simplest programs you will learn.

```
// C++ program to display "Hello World"
// Header file for input output functions
#include <iostream>
using namespace std;
```

```
// Main() function: where the execution of program begins
int main()
{
   // prints hello world
   cout << "Hello World";

   return 0;
}
```

Output
Hello World

Let us now understand every line and the terminologies of the above program:

1) // C++ program to display "Hello World": This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with '//' without quotes OR in between /*…*/ in C++ is comment.
2) #include: In C++, all lines that start with pound (#) sign are called directives and are processed by a preprocessor which is a program invoked by the compiler. The **#include** directive tells the compiler to include a file and **#include<iostream>**. It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.
3) using namespace std: This is used to import the entirety of the std namespace into the current namespace of the program.
4) int main(): This line is used to declare a function named "main" which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function.
5) { and }: The opening braces '{' indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function. Everything between these two comprises the body of the main function.
6) cout<<"Hello World";: This line tells the compiler to display the message "Hello World" on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ';' is used to end a statement. Semi-colon character at the end of the statement is used to indicate that the statement is ending there. Everything followed by the character "<<" is displayed to the output device
7) return 0; : This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.
8) Indentation: As you can see the cout and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance, but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you must always use indentations and comments to make the code more readable

C++ Basic Input/Output

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

There are following header files important to C++ programs −

Sr.No	Header File & Function and Description
1	**<iostream>** This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2	**<iomanip>** This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.
3	**<fstream>** This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

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

int main() {
  char str[] = "Hello C++";

  cout << "Value of str is : " << str << endl;
}
```

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

Value of str is : Hello C++

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
  char name[50];

  cout << "Please enter your name: ";
  cin >> name;
  cout << "Your name is: " << name << endl;

}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

Please enter your name: cplusplus
Your name is: cplusplus

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

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

int main() {
  char str[] = "Unable to read....";
```

```
  cerr << "Error message : " << str << endl;
}
```

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

Error message : Unable to read....

The Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

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

int main() {
  char str[] = "Unable to read....";

  clog << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result –
Error message : Unable to read....

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

Important Points to Note while Writing a C++ Program:

1. Always include the necessary header files for the smooth execution of functions. For example, **<iostream>** must be included to use **std::cin** and **std::cout**.
2. The execution of code begins from the **main()** function.
3. each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
4. It is a good practice to use **Indentation** and **comments** in programs for easy understanding.
5. **cout** is used to print statements and **cin** is used to take inputs.

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

Auto	Break	case	char	const	continue	default	do

Double	Else	enum	extern	float	for	goto	if
Int	Long	register	return	short	signed	sizeof	static
Struct	Switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

Asm	dynamic_cast	namespace	reinterpret_cast	bool
Explicit	New	static_cast	False	catch
Operator	Template	friend	Private	class
This	Inline	public	Throw	const_cast
Delete	Mutable	protected	True	try
Typeid	Typename	using	Virtual	wchar_t

C++ comments

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */. For example −

/* This is a comment */

/* C++ comments can also
 span multiple lines */
A comment can also start with //, extending to the end of the line. For example −

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

main() {
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

When the above code is compiled, it will ignore // **prints Hello World** and final executable will produce the following result −

Hello World

Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example −

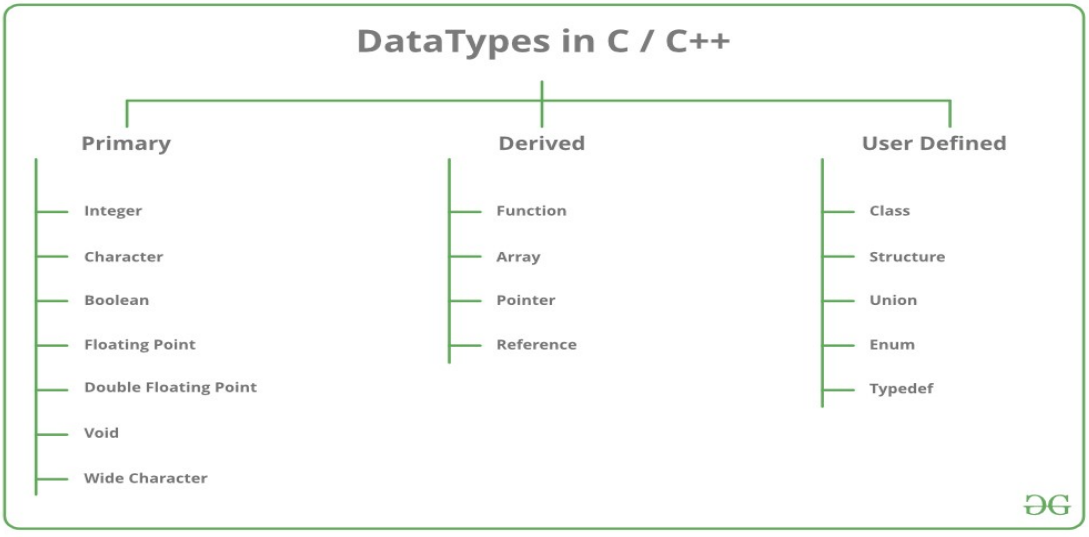
```
/* Comment out printing of Hello World:
cout << "Hello World"; // prints Hello World
*/
```

C++ Data Types

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory.
C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application. Data types specify the size and types of value to be stored. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines.

C++ supports the following data types:

1. Primary or Built in or Fundamental data type
2. Derived data types
3. User defined data types



Data types in C++ are mainly divided into three types:

1. Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean

- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

2. Derived Data Types: The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

3. Abstract or User-Defined Data Types: These data types are defined by the user itself. Like, as defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

- Class
- Structure
- Union
- Enumeration
- Typedef defined DataType

This article discusses **primitive data types** available in C++.

- **Integer**: The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. The keyword used for the boolean data type is **bool**.
- **Floating Point**: Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.
- **void**: Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

sizeof operator — sizeof operator is used to find the number of bytes occupied by a variable/data type in computer memory. Eg: int m , x[50]; cout<<sizeof(m); //returns 4 which is the number of bytes occupied by the integer variable “m”. cout<<sizeof(x); //returns 200 which is the number of bytes occupied by the integer array variable “x”.

// Following is the example, which will produce correct size of various data types on your

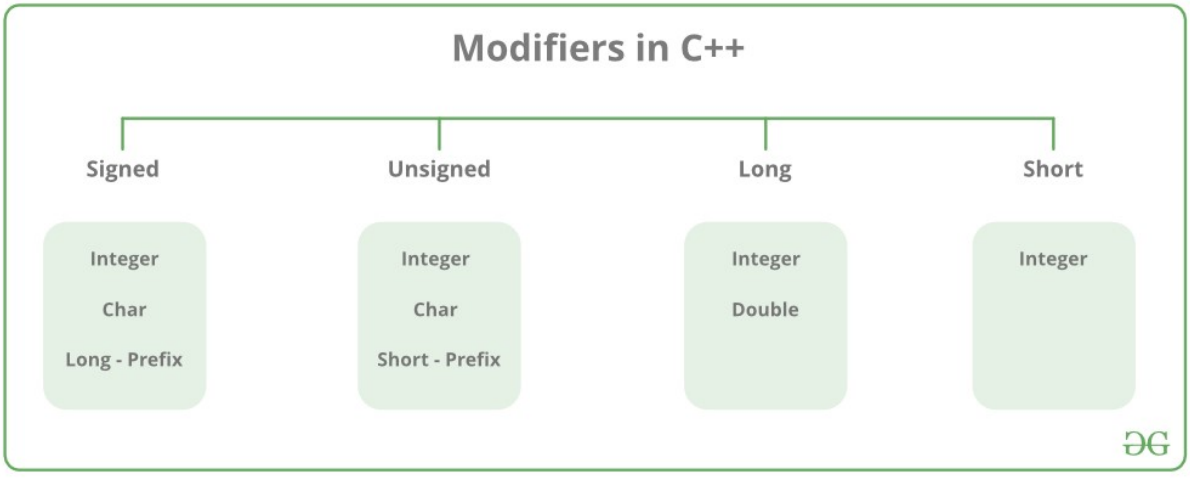
```
computer.
#include <iostream>
using namespace std;

int main()
{
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;

   cout << "Size of long : " << sizeof(long) << endl;
   cout << "Size of float : " << sizeof(float) << endl;

   cout << "Size of double : " << sizeof(double) << endl;

   return 0;
}
```

Output

Size of char : 1
Size of int : 4
Size of long : 8
Size of float : 4
Size of double : 8

Datatype Modifiers

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- **Signed**
- **Unsigned**
- **Short**
- **Long**

The below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
Int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
Float	4	
Double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Note: Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.

We can display the size of all the data types by using the sizeof() operator and passing the keyword of the datatype as an argument to this function as shown below:

Now to get the range of data types refer to the following chart

Note: syntax<limits.h> header file is defined to find the range of fundamental data-types. Unsigned modifiers have minimum value is zero. So, no macro constants are defined for the unsigned minimum value.

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers −

mohd zara abc move_name a_123 myname50 _temp j a23b9 retVal

Variable in C++

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.
Let's see the syntax to declare a variable:

```
type    variable_list;
```

The example of declaring variable is given below:

```
int x;
float y;
char z;
```

Here, x, y, z are variables and int, float, char are data types. We can also provide values while declaring the variables as given below:

```
int x=5,b=10;  //declaring 2 variable of integer type
float f=30.8;
char c='A';
```

Rules for defining variables

1. A variable can have alphabets, digits and underscore.
2. A variable name can start with alphabet and underscore only. It can't start with digit.
3. No white space is allowed within variable name.
4. A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int    a;
int   _ab;
int   a30;
```

Invalid variable names:

```
int 4;
int x y;
int double;
```

Variable Scope in C++

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

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

int main () {
  // Local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

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

// Global variable declaration:
int g;

int main () {
```

```
  // Local variable declaration:
  int a, b;

  // actual initialization
  a = 10;
  b = 20;
  g = a + b;

  cout << g;

  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example :

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

// Global variable declaration:
int g = 20;

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

  cout << g;

  return 0;
}
```

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

10

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initializer
Int	0
Char	'\0'
Float	0
Double	0
Pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

C++ Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1= 100 + 50;       // 150 (100 + 50)
int sum2= sum1 + 250;     // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

C++ divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values.

Note: The return value of a comparison is either true (1) or false (0).

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

Example

```
int x = 5;
int y = 3;
cout << (x > y); // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

Operator	Name	Example
==	Equal to	x == y

!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

You will learn much more about comparison operators and how to use them in a later chapter.

Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Predefined Math functions

C++ has many functions that allows you to perform mathematical tasks on numbers.

Max and min

The max(*x*,*y*) function can be used to find the highest value of *x* and *y*:

Example

```
cout << max(5, 10);
```

And the min(*x*,*y*) function can be used to find the lowest value of *x* and *y*:

Example

```
cout << min(5, 10);
```

C++ <cmath> Header

Other functions, such as sqrt (square root), round (rounds a number) and log (natural logarithm), can be found in the <cmath> header file:

Example

```
// Include the cmath library
#include <cmath>

cout << sqrt(64);
cout << round(2.6);
cout << log(2);
```

Other Math Functions

A list of other popular Math functions (from the <cmath> library) can be found in the table below:

Function	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x
asin(x)	Returns the arcsine of x
atan(x)	Returns the arctangent of x
cbrt(x)	Returns the cube root of x
ceil(x)	Returns the value of x rounded up to its nearest integer
cos(x)	Returns the cosine of x
cosh(x)	Returns the hyperbolic cosine of x
exp(x)	Returns the value of E^x
expm1(x)	Returns e^x -1
fabs(x)	Returns the absolute value of a floating x
fdim(x, y)	Returns the positive difference between x and y
floor(x)	Returns the value of x rounded down to its nearest integer
hypot(x, y)	Returns sqrt(x^2 +y^2) without intermediate overflow or underflow
fma(x, y, z)	Returns x*y+z without losing precision
fmax(x, y)	Returns the highest value of a floating x and y
fmin(x, y)	Returns the lowest value of a floating x and y
fmod(x, y)	Returns the floating point remainder of x/y
pow(x, y)	Returns the value of x to the power of y
sin(x)	Returns the sine of x (x is in radians)
sinh(x)	Returns the hyperbolic sine of a double value
tan(x)	Returns the tangent of an angle
tanh(x)	Returns the hyperbolic tangent of a double value

Manipulators in C++ with Examples

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

- Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.
- Manipulators are operators that are used to format the data display.
- To access manipulators, the file iomanip should be included in the program.

For example, if we want to print the hexadecimal value of 100 then we can print it as:

```
cout<<setbase(16)<<100
```

Types of Manipulators There are various types of manipulators:

1. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.
   - **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
   - **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
   - **ends**: It is also defined in ostream and it inserts a null character into the output stream.

Examples:

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
   istringstream str("          Programmer");
   string line;
   // Ignore all the whitespace in string
   // str before the first word.
   getline(str >> std::ws, line);

   // you can also write str>>ws
   // After printing the output it will automatically
   // write a new line in the output stream.
   cout << line << endl;

   // without flush, the output will be the same.
```

```
    cout << "only a test" << flush;

    // Use of ends Manipulator
    cout << "\na";

    // NULL character will be added in the Output
    cout << "b" << ends;
    cout << "c" << endl;

    return 0;
}
```

Output:
Programmer
only a test
abc

1. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill ('*'), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program. For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill ('*');
   - **Some important manipulators in <*iomanip*> are:**
     1. **setw (val):** It is used to set the field width in output operations.
     2. **setfill (character):** It is used to fill the character 'c' on output stream.
     3. **setprecision (val):** It sets val as the new value for the precision of floating-point values.
     4. **setbase(val):** It is used to set the numeric base value for numeric values.
     5. **setiosflags(flag):** It is used to set the format flags specified by parameter mask.

   - **Some important manipulators in <ios> are:**
     1. **showpos:** It forces to show a positive sign on positive numbers.
     2. **noshowpos:** It forces not to write a positive sign on positive numbers..
     3. **fixed:** It uses decimal notation for floating-point values.
     4. **scientific:** It uses scientific floating-point notation.
     5. **hex:** Read and write hexadecimal values for integers and it works same as the setbase(16).
     6. **dec:** Read and write decimal values for integers i.e. setbase(10).
     7. **oct:** Read and write octal values for integers i.e. setbase(10).
     8. **left:** It adjusts output to the left.
     9. **right:** It adjusts output to the right.

There are two types of manipulators used generally:

1] Parameterized and
2] Non-parameterized

1] Parameterized Manipulators:-

setw (n) -> To set field width to n
setprecision (p) -> The precision is fixed to p
setfill ('*') -> To set the character to be filled
setiosflags (l) -> Format flag is set to l
Setbase(b) -> To set the base of the number to b

- **setw() is a function in Manipulators in C++:**
  The setw() function is an output manipulator that inserts whitespace between two variables. You must enter an integer value equal to the needed space.

  Example:

```
int a=15;  int b=20;
cout << setw(10) << a << setw(10) << b << endl;
```

- **setfill() is a function in Manipulators in C++:**
  It replaces setw(whitespaces)'s with a different character. It's similar to setw() in that it manipulates output, but the only parameter required is a single character.

  Example:

```
int a,b;
a=15;   b=20;
cout<< setfill('*') << endl;
cout << setw(5) << a << setw(5) << b<< endl;
```

- **setprecision() is a function in Manipulators in C++:**
  It is an output manipulator that controls the number of digits to display after the decimal for a floating point integer.
  Example:

```
float A = 1.34255;
cout << setprecision(3) << A << endl;
```

- **setbase() is a function in Manipulators in C++:**
  The setbase() manipulator is used to change the base of a number to a different value. The following base values are supported by the C++ language:
  • hex (Hexadecimal = 16)
  • oct (Octal = 8)
  • dec (Decimal = 10)

The manipulators hex, oct, and dec can change the basis of input and output numbers.

```
// Example:

#include <iostream>
#include <iomanip>

using namespace std;
main()
{

int number = 100;
```

```
cout << "Hex Value =" << " " << hex << number << endl;
cout << "Octal Value=" << " " << oct << number << endl;
cout << "Setbase Value=" << " " << setbase(8) << number << endl;
cout << "Setbase Value=" << " " << setbase(16) << number << endl;

return 0;
}
```

Output

Hex Value = 64
Octal Value= 144
Setbase Value= 144
Setbase Value= 64

2] Non-parameterized

Examples are endl, fixed, showpoint

• endl – Gives a new line
• ends – Adds null character to close an output string
• hex, oct, dec – Displays the number in hexadecimal or octal or in decimal format

C++ decision making statements

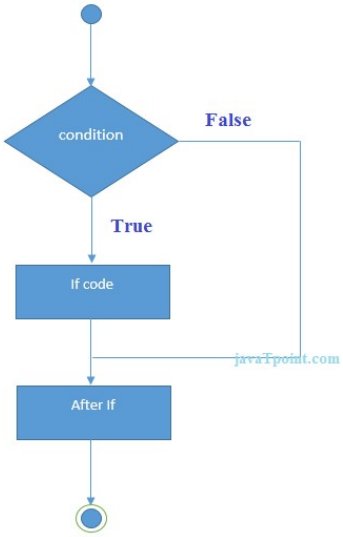
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

if Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
if(condition){
/code to be executed
}
```



Example:

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

int main ()
 {
   int num = 10;
         if (num % 2 == 0)
         {
            cout<<"It is even number";
         }
   return 0;
}
```

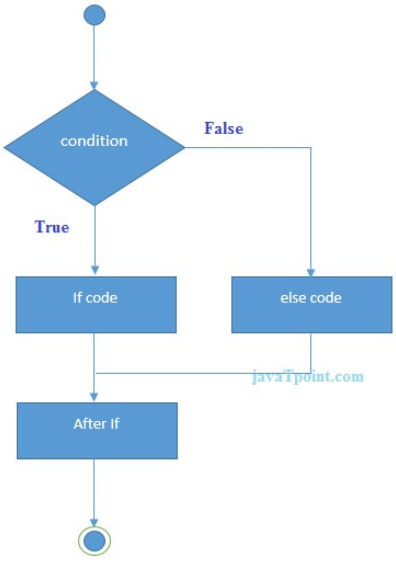
Output:

```
It is even number
```

IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



if-else Example

```
#include <iostream>
using namespace std;
int main () {
  int num = 11;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
        else
        {
           cout<<"It is odd number";
        }
  return 0;
}
```

Output:

```
It is odd number
```

if-else Example: with input from user

```
#include <iostream>
using namespace std;
int main () {
   int num;
   cout<<"Enter a Number: ";
   cin>>num;
        if (num % 2 == 0)
        {
           cout<<"It is even number"<<endl;
        }
        else
        {
           cout<<"It is odd number"<<endl;
        }
```

```
    return 0;
}
```

Output:

```
Enter a number:11
It is odd number
```

Output:

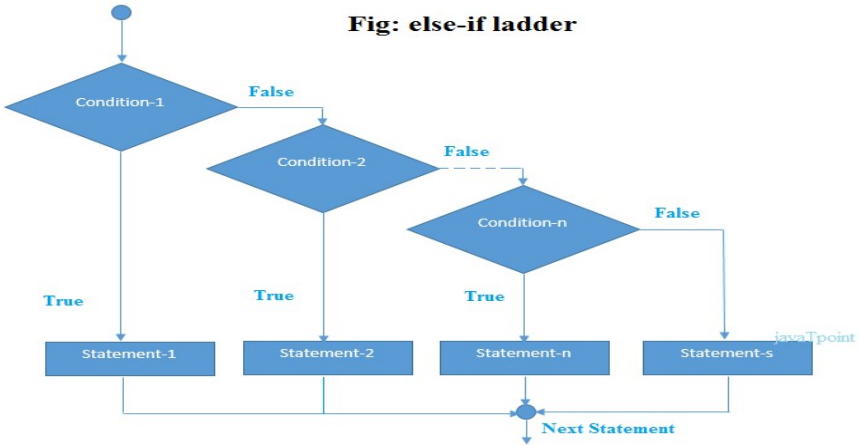
```
Enter a number:12
It is even number
```

if-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Fig: else-if ladder



if else-if Example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
        if (num <0 || num >100)
        {
            cout<<"wrong number";
        }
        else if(num >= 0 && num < 50){
            cout<<"Fail";
```

```
        }
        else if (num >= 50 && num < 60)
        {
           cout<<"D Grade";
        }
        else if (num >= 60 && num < 70)
        {
           cout<<"C Grade";
        }
        else if (num >= 70 && num < 80)
        {
           cout<<"B Grade";
        }
        else if (num >= 80 && num < 90)
        {
           cout<<"A Grade";
        }
        else if (num >= 90 && num <= 100)
        {
           cout<<"A+ Grade";
        }
    }
```

Output:

```
Enter a number to check grade:66
C Grade
```

Output:

```
Enter a number to check grade:-2
wrong number
```

Switch Statements

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break and default keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When C++ reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

Loops

In programming, sometimes there is a need to perform some operation **more than once** or (say) **n number** of times. Loops come into use when we need to repeatedly execute a block of statements.

For example: Suppose we want to print “Hello World” 10 times. This can be done in two ways as shown below:

Manual(general) Method (Iterative Method)

Manually we have to write cout statement 10 times. Let’s say you have to write it 20 times (it would surely take more time to write 20 statements) now imagine you have to write it 100 times, it would be really hectic to re-write the same statement again and again. So, here loops have their role.

```
// C++ program to illustrate need of loops
#include <iostream>
```

```
using namespace std;

int main()
{
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   cout << "Hello World\n";
   return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below. In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loop, the test condition is tested before entering the loop body. **For Loop** and **While Loop** is entry-controlled loops.
2. **Exit Controlled Loops**: In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. the do-while **loop** is exit controlled loop.

Loop Type and Description

1. **while loop** – First checks the condition, then executes the body.
2. **for loop** – firstly initializes, then, condition check, execute body, update.
3. **do-while** – firstly, execute the body then condition check

for Loop

A for loop is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one

line.

Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

Example:

```
for(int i = 0; i < n; i++){
}
```

In for loop, a loop variable is used to control the loop. First, initialize this loop variable to some value, then check whether this variable is less than or greater than the counter value. If the statement is true, then the loop body is executed and the loop variable gets updated. Steps are repeated till the exit condition comes.

- **Initialization Expression**: In this expression, we have to initialize the loop counter to some value. for example: int i=1;
- **Test Expression**: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression otherwise we will exit from the for a loop. For example: i <= 10;
- **Update Expression**: After executing the loop body this expression increments/decrements the loop variable by some value. for example: i++;
-

Example:

```
// C++ program to illustrate for loop
#include <iostream>
using namespace std;

int main()
{
   for (int i = 1; i <= 10; i++)
   {
      cout << "Hello World\n";
   }

   return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Hello World

While Loop

While studying **for loop** we have seen that the number of iterations is ***known beforehand***, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where **we do not know** the exact number of iterations of the loop **beforehand**. The loop execution is terminated on the basis of the test conditions.
Syntax: We have already stated that a loop mainly consists of three statements – initialization expression, test expression, and update expression. The syntax of the three loops – For, while, and do while mainly differs in the placement of these three statements.

```
initialization expression;
while (test_expression)
{
  // statements
 update_expression;
}
```

Example:

```
// C++ program to illustrate while loop
#include <iostream>
using namespace std;

int main()
{
   // initialization expression
   int i = 1;

   // test expression
   while (i < 6)
   {
      cout << "Hello World\n";

      // update expression
      i++;
   }

   return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

do-while loop

In do-while loops also the loop execution is terminated on the basis of test conditions. The main difference between a do-while loop and the while loop is in the do-while loop the condition is tested at the end of the loop body, i.e do-while loop is exit controlled whereas the other two loops are entry controlled loops.

Note: In a do-while loop, the loop body will ***execute at least once*** irrespective of the test condition.

Syntax:

```
initialization expression;
do
{
  // statements

  update_expression;
} while (test_expression);
```

Note: Notice the semi – colon(";") in the end of loop.

Example:

```
// C++ program to illustrate do-while loop
#include <iostream>
using namespace std;

int main()
{
   int i = 2; // Initialization expression

   do
   {
      // loop body
      cout << "Hello World\n";

      // update expression
      i++;

   }  while (i < 1);   // test expression

   return 0;
}
```

Output:

Hello World

In the above program, the test condition (i<1) evaluates to false. But still, as the loop is an exit – controlled the loop body will execute once.

Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a **functional exit** so that it repeats indefinitely. An infinite loop occurs when a condition is

always evaluated to be true. Usually, this is an error.

Using For loop:

```
// C++ program to demonstrate infinite loops using for
#include <iostream>
using namespace std;
int main ()
{
   int i;

   // This is an infinite for loop as the condition
   // expression is blank
   for ( ; ; )
   {
      cout << "This loop will run forever.\n";
   }
}
```

Output:

This loop will run forever.
This loop will run forever.
.....

Using While loop:

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

int main()
{

   while (1)
      cout << "This loop will run forever.\n";
   return 0;
}
```

Output:

This loop will run forever.
This loop will run forever.
.....

Using Do-While loop:

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

int main() {
```

```
    do{
       cout << "This loop will run forever.\n";
    } while(1);

    return 0;
}
```

Output:

This loop will run forever.
This loop will run forever.
.....

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.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

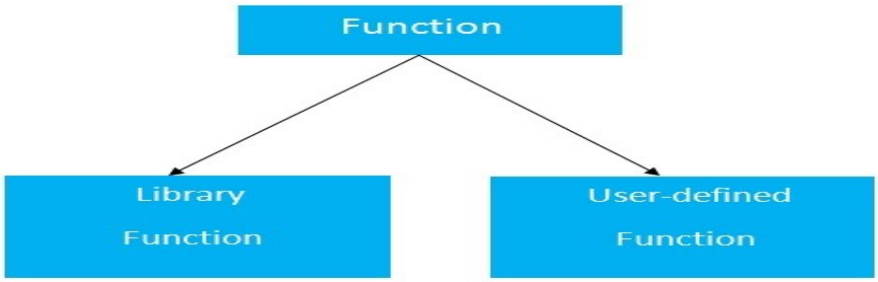
Advantage of functions

- There are many advantages of functions.
- **1) Code Reusability**
- By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.
- **2) Code optimization**
- It makes the code optimized, we don't need to write much code.
- Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.
- But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

- **1. Library Functions:** are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.
- **2. User-defined functions:** are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

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 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.

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**:

Example

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

void myFunction() {
```

```
  cout << "I just got executed!";
}

// Error
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read:

Example

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

Example

```
void myFunction() {
  cout << "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
```

```
 return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Example:
Below is a simple program to demonstrate functions.

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

int max(int x, int y)
{
   if (x > y)
   return x;
   else
   return y;
}

int main() {
   int a = 10, b = 20;

   // Calling above function to find max of 'a' and 'b'
   int m = max(a, b);

   cout << "m is " << m;
   return 0;
}
```

Output:
m is 20

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.
The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.
There are two most popular ways to pass parameters.
Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.
Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.
Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

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

void fun(int x) {
```

```
    x = 30;
}

int main() {
    int x = 20;
    fun(x);
    cout << "x = " << x;
    return 0;
}
```

Output:
x = 20

However, in C, we can use pointers to get the effect of pass-by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement '*ptr = 30', value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&x)', the address of x is passed so that x can be modified using its address.

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

void fun(int *ptr)
{
    *ptr = 30;
}

int main() {
    int x = 20;
    fun(&x);
    cout << "x = " << x;

    return 0;
}
```

Output:
x = 30

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

Following are some important points about functions in C++.

1) Every C++ program has a function called main() that is called by operating system when a user runs the program.

2) Every function has a return type. If a function doesn't return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolize the termination of the function as shown below:

```
void function name(int a)
{
.....  //Function Body
return;  //Function execution would get terminated
}
```

3) In C and C++, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

4) Empty parameter list means that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use "void fun(void)". As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

Main Function:

The main function is a special function. Every C++ program must contain a function named main.

It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

Types of main Function:

1) The first type is – main function without parameters :

```
// Without Parameters
int main()
{
  ...
  return 0;
}
```

2) Second type is main function with parameters :

```
// With Parameters
int main(int argc, char * const argv[])
{
  ...
  return 0;
}
```

The reason for having the parameter option for the main function is to allow input from the command line.

When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named argv.

Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

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

Storage Class	Keyword	Lifetime	Visibility	Initial Value

Automatic	Auto	Function Block	Local	Garbage
Register	Register	Function Block	Local	Garbage
External	Extern	Whole Program	Global	Zero
Static	Static	Whole Program	Local	Zero

Automatic Storage Class

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

```
{
auto int y;
float y = 3.45;
}
```

The above example defines two variables with a same storage class, auto can only be used within functions.

Register Storage Class

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables. It is recommended to use register variable only for quick access such as in counter. We can't get the address of register variable.

```
register int counter=0;
```

Static Storage Class

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

```
#include <iostream>
using namespace std;
void func() {
  static int i=0; //static variable
  int j=0; //local variable
  i++;
  j++;
  cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
```

```
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

External Storage Class

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

```
extern int counter=0;
```

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ std::array is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.

Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.
- Disadvantages of C++ Array
- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
       //traversing array
```

```
        for (int i = 0; i < 5; i++)
        {
            cout<<arr[i]<<"\n";
        }
    }
```

Output:

```
10
0
20
0
30
```

C++ Array Example: Traversal using for each loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
      //traversing array
      for (int i: arr)
      {
          cout<<i<<"\n";
      }
}
```

Output:

```
10
20
30
40
50
```

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
      int arr1[5] = { 10, 20, 30, 40, 50 };
      int arr2[5] = { 5, 15, 25, 35, 45 };
```

```
        printArray(arr1); //passing array to function
        printArray(arr2);
}
void printArray(int arr[5])
{
   cout << "Printing array elements:"<< endl;
   for (int i = 0; i < 5; i++)
   {
            cout<<arr[i]<<"\n";
   }
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Passing Array to Function Example 1:

Let's see an example of C++ array which prints minimum number in an array using function.

```
#include <iostream>
using namespace std;
void  printMin(int arr[5]);
int main()
{
  int arr1[5] = { 30, 10, 20, 40, 50 };
      int arr2[5] = { 5, 15, 25, 35, 45 };
      printMin(arr1);//passing array to function
       printMin(arr2);
}
void  printMin(int arr[5])
{
   int min = arr[0];
      for (int i = 0; i > 5; i++)
      {
         if (min > arr[i])
         {
            min = arr[i];
         }
```

```
        }
        cout<< "Minimum element is: "<< min <<"\n";
    }
```

Output:

Minimum element is: 10
Minimum element is: 5

C++ Passing Array to Function Example 2:

Let's see an example of C++ array which prints maximum number in an array using function.

```
#include <iostream>
using namespace std;
void  printMax(int arr[5]);
int main()
{
    int arr1[5] = { 25, 10, 54, 15, 40 };
    int arr2[5] = { 12, 23, 44, 67, 54 };
    printMax(arr1); //Passing array to function
     printMax(arr2);
}
void  printMax(int arr[5])
{
   int max = arr[0];
    for (int i = 0; i < 5; i++)
    {
       if (max < arr[i])
       {
          max = arr[i];
       }
    }
    cout<< "Maximum element is: "<< max <<"\n";
}
```

Output:

Maximum element is: 54
Maximum element is: 67

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} , {4, 5, 6, 7} , {8, 9, 10, 11}};

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 4^{th} element from the 3^{rd} 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.

Return Array from Functions

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 generate 10 random numbers and return them using an array and call this function as follows −

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

using namespace std;

// function to generate and retrun random numbers.
int * getRandom( ) {

  static int  r[10];

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

  for (int i = 0; i < 10; ++i) {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

// main function to call above defined function.
int main () {

  // a pointer to an int.
  int *p;

  p = getRandom();

  for ( int i = 0; i < 10; i++ ) {
    cout << "*(p + " << i << ") : ";
    cout << *(p + i) << endl;
  }

  return 0;
}
```

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

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
```

```
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

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**.

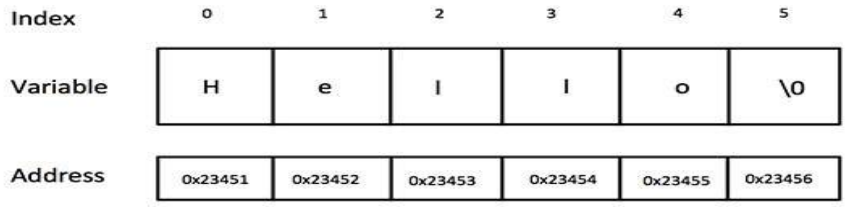
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
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++ Pointers

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a name of memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined −

```
#include <iostream>

using namespace std;
int main () {
  int  var1;
  char var2[10];

  cout << "Address of var1 variable: ";
  cout << &var1 << endl;

  cout << "Address of var2 variable: ";
  cout << &var2 << endl;

  return 0;
}
```

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

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

```
int  *ip; // pointer to an integer
double  *dp; // pointer to a double
float  *fp; // pointer to a float
char  *ch // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently.

(a) We define a pointer variable.

(b) Assign the address of a variable to a pointer.

(c) Finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

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

int main () {
  int  var = 20;   // actual variable declaration.
  int  *ip;        // pointer variable
  ip = &var;       // store address of var in pointer variable

  cout << "Value of var variable: ";
  cout << var << endl;

  // print the address stored in ip pointer variable
  cout << "Address stored in ip variable: ";
  cout << ip << endl;

  // access the value at the address available in pointer
  cout << "Value of *ip variable: ";
  cout << *ip << endl;

  return 0;
}
```

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

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of *ip variable: 20

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
```

```
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
  return 0;
}
```

Output:

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

Pointers vs Arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have array address in pointer.
  ptr = var;

  for (int i = 0; i < MAX; i++) {
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;

    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;

    // point to the next location
    ptr++;
  }

  return 0;
}
```

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

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

However, pointers and arrays are not completely interchangeable. For example, consider the following program −

```
#include <iostream>
```

```
using namespace std;
const int MAX = 3;

int main () {
  int  var[MAX] = {10, 100, 200};

  for (int i = 0; i < MAX; i++) {
    *var = i;    // This is a correct syntax
    var++;       // This is incorrect.
  }

  return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value.

Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500 −

*(var + 2) = 500;

Above statement is valid and will compile successfully because var is not changed.

Passing Pointers to Functions

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

```
#include <iostream>
#include <ctime>
using namespace std;
void getSeconds(unsigned long *par);

int main () {
  unsigned long sec;
  getSeconds( &sec );

  // print the actual value
  cout << "Number of seconds :" << sec << endl;

  return 0;
}

void getSeconds(unsigned long *par) {
  // get the current number of seconds
  *par = time( NULL );
```

```
   return;
}
```

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

Number of seconds :1294450468

The function which can accept a pointer, can also accept an array as shown in the following example −

```
#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;
}

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;
}
```

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

Average value is: 214.4

C++ structures

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title

- Author
- Subject
- Book ID
-

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

```
struct [structure tag] {
member definition;
member definition;
...
member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
      char title[50];
      char author[50];
      char subject[100];
      int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure −

```
#include <iostream>
#include <cstring>

using namespace std;

struct  Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main() {
  struct  Books Book1;       // Declare Book1 of type Book
  struct  Books Book2;       // Declare Book2 of type Book

  // book 1 specification
  strcpy( Book1.title, "Learn C++ Programming");
  strcpy( Book1.author, "Chand Miyan");
  strcpy( Book1.subject, "C++ Programming");
```

```
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;

   return 0;
}
```

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

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example −

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book
```

```
   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   printBook( Book1 );
   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book ) {
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

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

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
int main() {
   struct Books  Book1;        // Declare Book1 of type Book
   struct Books  Book2;        // Declare Book2 of type Book

   // Book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

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

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
```

Book id : 6495700

The typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example −

```
typedef struct {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Books;
```

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example −

```
Books Book1, Book2;
```

You can use **typedef** keyword for non-structs as well as follows −

```
typedef  long  int   *pint32;
pint32   x, y, z;
```

x, y and z are all pointers to long ints.

Classes and Objects

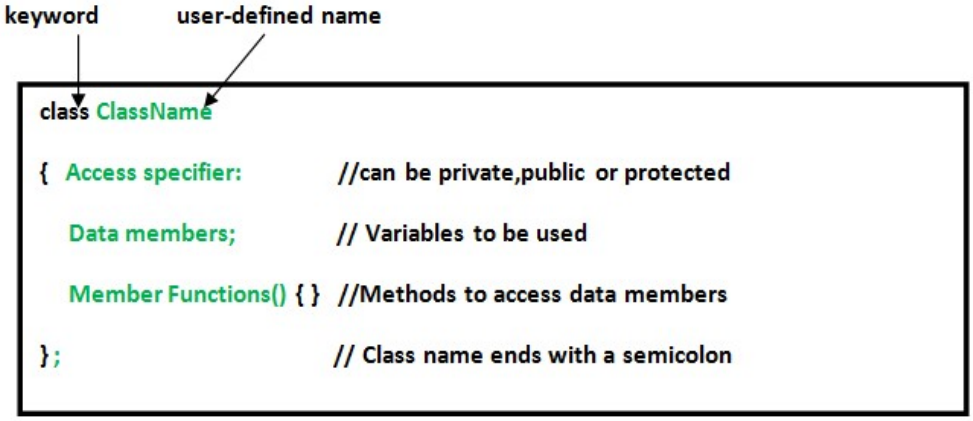
Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

```
// C++ program to demonstrate
// accessing of data members

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
   // Access specifier
   public:

   // Data Members
   string geekname;

   // Member Functions()
   void printname()
   {
```

```
        cout << "Geekname is: " << geekname;
    }
};

int main() {

    // Declare an object of class geeks
    Geeks obj1;

    // accessing data member
    obj1.geekname = "Abhi";

    // accessing member function
    obj1.printname();
    return 0;
}
```

Output:
Geekname is: Abhi

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
// C++ program to demonstrate function
// declaration outside class

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    string geekname;
    int id;

    // printname is not defined inside class definition
    void printname();

    // printid is defined inside class definition
    void printid()
    {
        cout << "Geek id is: " << id;
    }
};

// Definition of printname using scope resolution operator ::
```

```
void Geeks::printname()
{
   cout << "Geekname is: " << geekname;
}
int main() {

   Geeks obj1;
   obj1.geekname = "xyz";
   obj1.id=15;

   // call printname()
   obj1.printname();
   cout << endl;

   // call printid()
   obj1.printid();
   return 0;
}
```

Output:
Geekname is: xyz

Geek id is: 15

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.
Note: Declaring a friend function is a way to give private access to a non-member function.
Constructors
Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.
There are 3 types of constructors:

- Default constructors
- Parameterized constructors
- Copy constructors

```
// C++ program to demonstrate constructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
   public:
   int id;

   //Default Constructor
   Geeks()
```

```
    {
        cout << "Default Constructor called" << endl;
        id=-1;
    }

    //Parameterized Constructor
    Geeks(int x)
    {
        cout << "Parameterized Constructor called" << endl;
        id=x;
    }
};
int main() {

    // obj1 will call Default Constructor
    Geeks obj1;
    cout << "Geek id is: " <<obj1.id << endl;

    // obj1 will call Parameterized Constructor
    Geeks obj2(21);
    cout << "Geek id is: " <<obj2.id << endl;
    return 0;
}
```

Output:

Default Constructor called

Geek id is: -1

Parameterized Constructor called

Geek id is: 21

A **Copy Constructor** creates a new object, which is exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.

Syntax:

class-name (class-name &){}

Destructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

```
// C++ program to explain destructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    int id;

    //Definition for Destructor
    ~Geeks()
```

```
    {
        cout << "Destructor called for id: " << id <<endl;
    }
};

int main()
  {
    Geeks obj1;
    obj1.id=7;
    int i = 0;
    while ( i < 5 )
    {
        Geeks obj2;
        obj2.id=i;
        i++;
    } // Scope for obj2 ends here

    return 0;
  } // Scope for obj1 ends here
```

Output:

Destructor called for id: 0

Destructor called for id: 1

Destructor called for id: 2

Destructor called for id: 3

Destructor called for id: 4

Destructor called for id: 7

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming