Introduction to C++ Programming Language

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a **compiled** language.

C++ is a **middle-level language** rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). 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 libray 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 grabage-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 ca be built. Procedural code becomes difficult to maintain as code-size grows.
- **Compiled Language**: C++ is a compiled language, contributing to its speed.

1.1 Procedural Vs Object Oriented Programming

Procedural Programming

Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Languages used in Procedural Programming

```
FORTRAN, ALGOL, COBOL,
BASIC, Pascal and C.
```

Object Oriented Programming

Object oriented programming can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Languages used in Object Oriented Programming

```
Java, C++, C#, Python,
PHP, JavaScript, Ruby, Perl,
Objective-C, Dart, Swift, Scala.
```

Following are the important differences between OOP and POP.

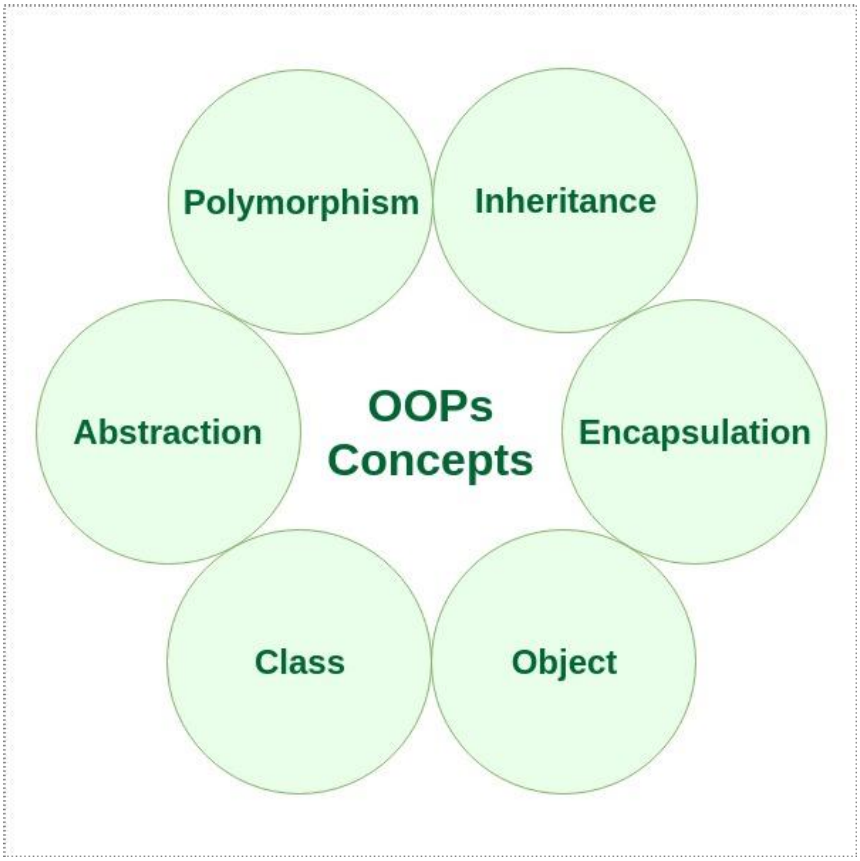
Sr. No.	Key	OOP	POP
1	**Definition**	OOP stands for Object Oriented Programming.	POP stands for Procedural Oriented Programming.
2	**Approach**	OOP follows bottom up approach.	POP follows top down approach.
3	**Division**	A program is divided to objects and their interactions.	A program is divided into functions and they interacts.
4	**Inheritance supported**	Inheritance is supported.	Inheritance is not supported.
5	**Access control**	Access control is supported via access modifiers.	No access modifiers are supported.
6	**Data Hiding**	Encapsulation is used to hide data.	No data hiding present. Data is globally accessible.
7	**Example**	C++, Java	C, Pascal

1.2 Basic Concepts of Object Oriented Programming

1. **Introduction**
2. **Class**
3. **Objects**
4. **Encapsulation**
5. **Abstraction**
6. **Polymorphism**
7. **Inheritance**
8. **Dynamic Binding**
9. **Message Passing**

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming



Class

The building block of C++ that leads to Object-Oriented programming is a Class. 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 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 define the properties and behaviour 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 apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object

An Object is an identifiable entity with some characteristics and behaviour. 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.

```
class person
{
char name[20];
int id;
public:
void getdetails(){}
};

int main()
{
person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

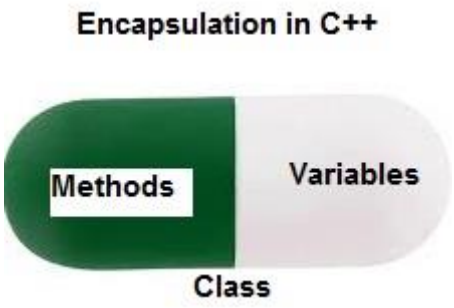
When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.
Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation in C++

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access

specifiers. A Class can decide which data member will be visible to the outside world and which is not.

- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Polymorphism

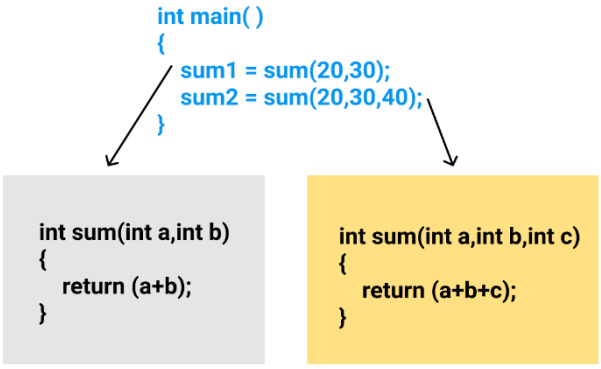
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



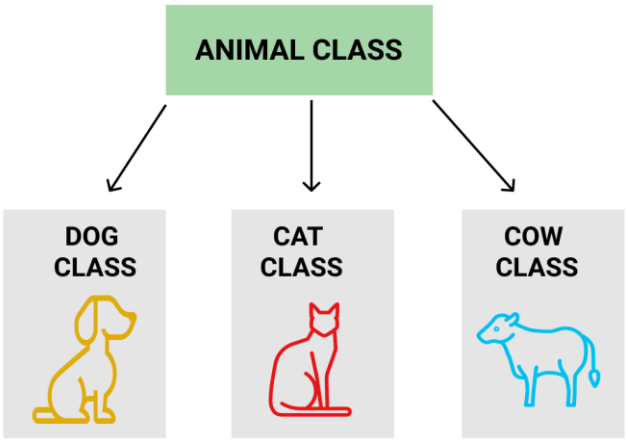
Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we

want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding

In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Applications of Object Oriented Programming

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modelling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

1.3 Structure of C++ Program

The structure of C++ program is divided into four different sections:
(1) Header File Section
(2) Class Declaration section

(3) Member Function definition section
(4) Main function section

(1) Header File Section :
This section contains various header files.
You can include various header files in to your program using this section.

For example:
```
# include <iostream.h >
```

Header file contains declaration and definition of various built in functions as well as object. In order to use this built in functions or object we need to include particular header file in our program.

(2) Class Declaration Section:
This section contains declaration of class.
You can declare class and then declare data members and member functions inside that class.
For example:
```
class Demo
{
int a, b;
public:
void input();
void output();
}
```

You can also inherit one class from another existing class in this section.

(3) Member Function Definition Section:
This section is optional in the structure of C++ program. Because you can define member functions inside the class or outside the class. If all the member functions are defined inside the class then there is no need of this section.
This section is used only when you want to define member function outside the class.
This section contains definition of the member functions that are declared inside the class.

For example:
```
void Demo:: input ()
{
cout << “Enter Value of A:”;
cin >> a;
cout << “Enter Value of B:”;
cin >> b;
}
```

(4) Main Function Section:
o In this section you can create an object of the class and then using this object you can call various functions defined inside the class as per your requirement.

```
For example:
Void main ()
{
Demo d1;
d1.input ();
d1.output ();
}
```

We can also compare the structure of C++ program with client server application. In client server application client send request to the server and server sends response to the client.
In above C++ structure the class declaration section and member function definition section both together works as a server and main () function section works as a client. Because in main () function section we create an object of the class and then using that object we make a call to the function declared in the class.

1.4 Tokens

Each word and punctuation is referred to as a token in C++. Tokens are the smallest building block or smallest unit of a C++ program.
These following tokens are available in C++:

- **Identifiers**
- **Keywords**
- **Constants**
- **Operators**
- **Strings**

Identifiers

Identifiers are names given to different entries such as variables, structures, and functions. Also, identifier names should have to be unique because these entities are used in the execution of the program.

Keywords

Keywords are reserved words which have fixed meaning, and its meaning cannot be changed. The meaning and working of these keywords are already known to the compiler. C++ has more numbers of keyword than C, and those extra ones have special working capabilities.

Operators

C++ operator is a symbol that is used to perform mathematical or logical manipulations.

Constants

Constants are like a variable, except that their value never changes during execution once defined.

Strings

Strings are objects that signify sequences of characters.

1.5 Variables

Variables play a significant role in constructing a program, storing values in memory and dealing with them. Variables are required in various functions of every program. For example, when we check for conditions to execute a block of statements, variables are required. Again for iterating or repeating a block of the statement(s) several times, a counter variable is set along with a condition, or simply if we store the age of an employee, we need an integer type variable. So in every respect, the variable is used. In this tutorial, you will learn about how variables are declared in C++, how to assign values to them, and how to use them within a C++ program.

What are Variables?

Variables are used in C++ where you will need to store any type of values within a program and whose value can be changed during the program execution. These variables can be declared in various ways each having different memory requirements and storing capability. Variables are the name of memory locations that are allocated by compilers, and the allocation is done based on the data type used for declaring the variable.

Variable Definition in C++

A variable definition means that the programmer writes some instructions to tell the compiler to create the storage in a memory location. The syntax for defining variables is:

Syntax:

```
data_type variable_name;
data_type variable_name, variable_name, variable_name;
```

Here *data_type* means the valid C++ data type which includes int, float, double, char, wchar_t, bool and variable list is the lists of variable names to be declared which is separated by commas.

Example:

```
/* variable definition */int    width, height, age;
char   letter;
float  area;
double d;
```

Variable Initialization in C++

Variables are declared in the above example, but none of them has been assigned any value. Variables can be initialized, and the initial value can be assigned along with their declaration.

Syntax:

```
data_type variable_name = value;
```

Example:

```
/* variable definition and initialization */int    width, height=5, a
ge=32;
char   letter='A';
float  area;
double d;

/* actual initialization */width = 10;
area = 26.5;
```

There are some rules that must be in your knowledge to work with C++ variables.

Rules of Declaring variables in C++

- A variable name can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.
- The first character must be a letter or underscore.
- Blank spaces cannot be used in variable names.
- Special characters like #, $ are not allowed.
- C++ keywords cannot be used as variable names.
- Variable names are case-sensitive.
- A variable name can be consisting of 31 characters only if we declare a variable more than one characters compiler will ignore after 31 characters.
- Variable type can be bool, char, int, float, double, void or wchar_t.

Here's a Program to Show the Usage of Variables in C++

Example:

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

int main()
{
    int x = 5;
    int y = 2;
    int Result;
    Result = x * y;
    cout << Result;
}
```

Another program showing how Global variables are declared and used within a program:

Example:

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

// Global Variable declaration:
int x, y;
float f;

int main()
{
    // Local variable
    int tot;
    float f;
```

```
    x = 10;
    y = 20;
    tot = x + y;

    cout << tot;
    cout << endl;
    f = 70.0 / 3.0;
    cout << f;
    cout << endl;
}
```

1.6 Data Types

Data types in any of the language mean that what are the various type of data the variables can have in that particular language. Information is stored in computer memory with different data types. Whenever a variable is declared it becomes necessary to define a data type that what will be the type of data that variable can hold.

Data Types available in C++:

1. **Primary(Built-in) Data Types:**
    - **character**
    - **integer**
    - **floating point**
    - **boolean**
    - **double floating point**
    - **void**
    - **wide character**
2. **User Defined Data Types:**
    - **Structure**
    - **Union**
    - **Class**
    - **Enumeration**
3. **Derived Data Types:**
    - **Array**
    - **Function**
    - **Pointer**
    - **Reference**

Character Data Types

Data Type (Keywords)	Description	Size	Typical Range
char	Any single character. It may include a letter, a digit, a punctuation mark, or a space.	1 byte	-128 to 127 or 0 to 255

signed char	Signed character.	1 byte	-128 to 127
unsigned char	Unsigned character.	1 byte	0 to 255
wchar_t	Wide character.	2 or 4 bytes	1 wide character

Integer Data Types

Data Type (Keywords)	Description	Size	Typical Range
int	Integer.	4 bytes	-2147483648 to 2147483647
signed int	Signed integer. Values may be negative, positive, or zero.	4 bytes	-2147483648 to 2147483647
unsigned int	Unsigned integer. Values are always positive or zero. Never negative.	4 bytes	0 to 4294967295
short	Short integer.	2 bytes	-32768 to 32767
signed short	Signed short integer. Values may be negative, positive, or zero.	2 bytes	-32768 to 32767
unsigned short	Unsigned short integer. Values are always positive or zero. Never negative.	2 bytes	0 to 65535
long	Long integer.	4 bytes	-2147483648 to 2147483647
signed long	Signed long integer. Values may be negative, positive, or zero.	4 bytes	-2147483648 to 2147483647
unsigned long	Unsigned long integer. Values are always positive or zero. Never negative.	4 bytes	0 to 4294967295

Floating-point Data Types

Data Type (Keywords)	Description	Size	Typical Range
float	Floating point number. There is no fixed number of digits before or after the decimal point.	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number. More accurate compared to float.	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8 bytes	+/- 1.7e +/- 308 (~15 digits)

Boolean Data Type

Data Type (Keywords)	Description	Size	Typical Range
bool	Boolean value. It can only take one of two values: true or false.	1 byte	true or false

Variables sizes might be different in your PC from those shown in the above table, depending on the compiler you are using.

Below example will produce the correct size of various data type, on your computer.

Example:

```
#include <iostream>
using namespace std;
int main() {
 cout << "Size of char is " << sizeof(char) << endl;
 cout << "Size of int is " << sizeof(int) << endl;
 cout << "Size of float is " << sizeof(float) << endl;
 cout << "Size of short int is " << sizeof(short int) << endl;
 cout << "Size of long int is " << sizeof(long int) << endl;
 cout << "Size of double is " << sizeof(double) << endl;
 cout << "Size of wchar_t is " << sizeof(wchar_t) << endl;
 return 0;
}
```

Program Output:

```
Size of char is 1
Size of int is 4
Size of float is 4
Size of short int is 2
Size of long int is 4
Size of double is 8
Size of wchar_t is 4
```

Enum Data Type

This is a user-defined data type having a finite set of enumeration constants. The keyword 'enum' is used to create an enumerated data type.

Syntax:

```
enum enum-name {list of names}var-list;
enum mca(software, internet, seo);
```

Typedef

It is used to create a new data type. But it is commonly used to change the existing data type with another name.

Syntax:

```
typedef [data_type] synonym;
```

or

```
typedef [data_type] new_data_type;
```

Example:

```
typedef int integer;
integer rollno;
```

1.7 Type Casting

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator.

The most general cast supported by most of the C++ compilers is as follows −

```
(type) expression
```

Where type is the desired data type. There are other casting operators supported by C++, they are listed below −

- **const_cast<type> (expr)** − The const_cast operator is used to explicitly override const and/or volatile in a cast. The target type must be the same as the source type except for the alteration of its const or volatile attributes. This type of casting manipulates the const attribute of the passed object, either to be set or removed.
- **dynamic_cast<type> (expr)** − The dynamic_cast performs a runtime cast that verifies the validity of the cast. If the cast cannot be made, the cast fails and the expression evaluates to null. A dynamic_cast performs casts on polymorphic types and can cast a A* pointer into a B* pointer only if the object being pointed to actually is a B object.
- **reinterpret_cast<type> (expr)** − The reinterpret_cast operator changes a pointer to any other type of pointer. It also allows casting from pointer to an integer type and vice versa.
- **static_cast<type> (expr)** − The static_cast operator performs a nonpolymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer.

All of the above-mentioned casting operators will be used while working with classes and objects. For now, try the following example to understand a simple cast operators available in C++. Copy and paste the following C++ program in test.cpp file and compile and run this program.

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

main() {
   double a = 21.09399;
   float b = 10.20;
   int c ;

   c = (int) a;
   cout << "Line 1 - Value of (int)a is :" << c << endl ;

   c = (int) b;
   cout << "Line 2 - Value of (int)b is  :" << c << endl ;
```

```
   return 0;
}
```

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

```
Line 1 - Value of (int)a is :21
Line 2 - Value of (int)b is  :10
```

1.8 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language −

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.
Assume variable A holds 1 and variable B holds 0, then −
Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011
The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C++ language −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A

%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators

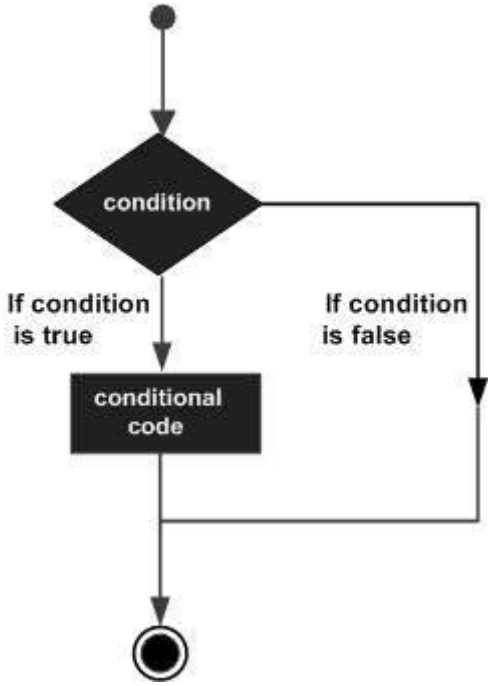
The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	*** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

1.9 Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C++ programming language provides following types of decision making statements.

Sr.No	Statement & Description
1	if statement An 'if' statement consists of a boolean expression followed by one or more statements.
2	if...else statement An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.

3	switch statement A 'switch' statement allows a variable to be tested for equality against a list of values.
4	nested if statements You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	nested switch statements You can use one 'switch' statement inside another 'switch' statement(s).

The ? : Conditional Operator

We have covered conditional operator "? :" in previous chapter which can be used to replace **if...else** statements. It has the following general form −

```
Exp1 ? Exp2 : Exp3;
```

Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a '?' expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire '?' expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

1. if statement

An **if** statement consists of a boolean expression followed by one or more statements.

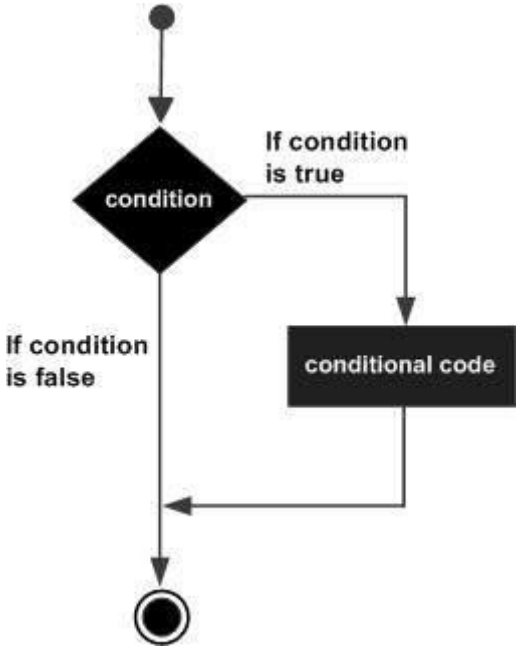
Syntax

The syntax of an if statement in C++ is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

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

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

   // check the boolean condition
   if( a < 20 ) {
      // if condition is true then print the following
      cout << "a is less than 20;" << endl;
   }
   cout << "value of a is : " << a << endl;

   return 0;
}
```

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

```
a is less than 20;
value of a is : 10
```

2. if...else statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

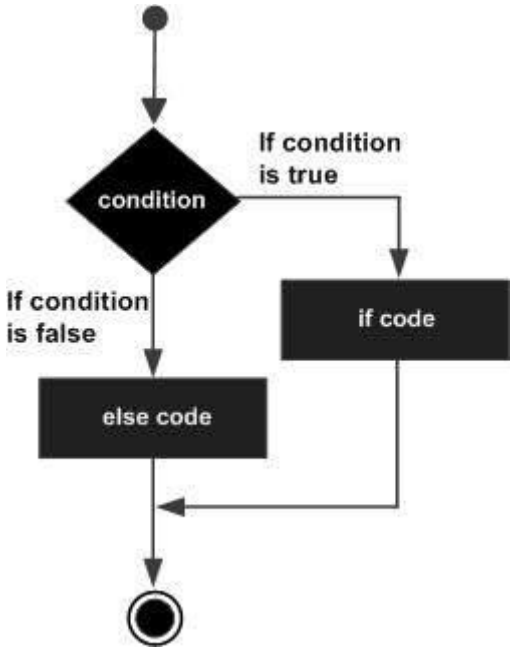
Syntax

The syntax of an if...else statement in C++ is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true
} else {
  // statement(s) will execute if the boolean expression is false
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



Example

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

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

   // check the boolean condition
   if( a < 20 ) {
      // if condition is true then print the following
      cout << "a is less than 20;" << endl;
   } else {
      // if condition is false then print the following
      cout << "a is not less than 20;" << endl;
   }
   cout << "value of a is : " << a << endl;

   return 0;
```

```
}
```

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

```
a is not less than 20;
value of a is : 100
```

if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C++ is −

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true
} else {
   // executes when the none of the above condition is true.
}
```

Example

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

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

   // check the boolean condition
   if( a == 10 ) {
      // if condition is true then print the following
      cout << "Value of a is 10" << endl;
   } else if( a == 20 ) {
      // if else if condition is true
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      // if else if condition is true
      cout << "Value of a is 30" << endl;
   } else {
      // if none of the conditions is true
      cout << "Value of a is not matching" << endl;
   }
```

```
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

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

```
Value of a is not matching
Exact value of a is : 100
```

3. Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax for a **switch** statement in C++ is as follows −

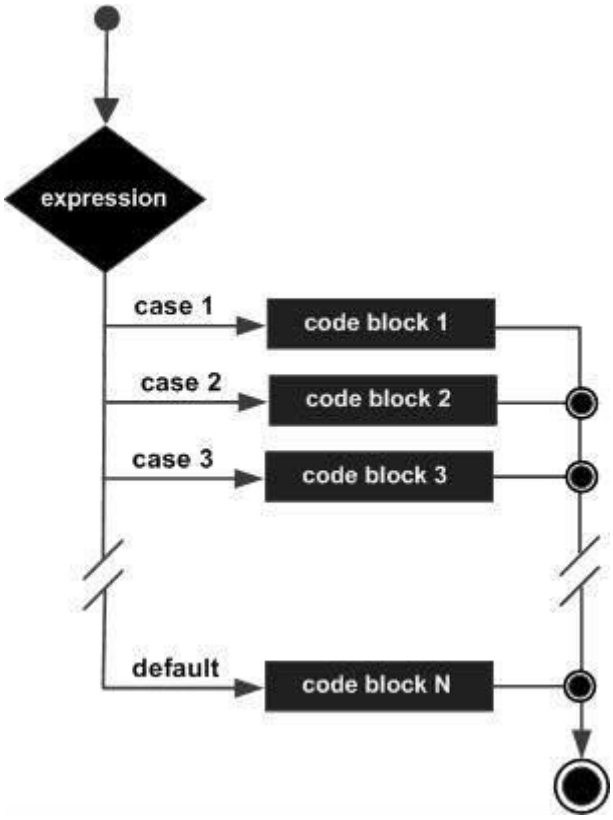
```
switch(expression) {
   case constant-expression  :
      statement(s);
      break; //optional
   case constant-expression  :
      statement(s);
      break; //optional

   // you can have any number of case statements.
   default : //Optional
      statement(s);
}
```

The following rules apply to a switch statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

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

int main () {
   // local variable declaration:
   char grade = 'D';

   switch(grade) {
      case 'A' :
         cout << "Excellent!" << endl;
         break;
      case 'B' :
      case 'C' :
         cout << "Well done" << endl;
         break;
      case 'D' :
         cout << "You passed" << endl;
         break;
      case 'F' :
         cout << "Better try again" << endl;
         break;
```

```
      default :
         cout << "Invalid grade" << endl;
   }
   cout << "Your grade is " << grade << endl;

   return 0;
}
```

This would produce the following result −

```
You passed
Your grade is D
```

4. Nested if statements

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows −

```
if( boolean_expression 1) {
   // Executes when the boolean expression 1 is true
   if(boolean_expression 2) {
      // Executes when the boolean expression 2 is true
   }
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

Example

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

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

   // check the boolean condition
   if( a == 100 ) {
      // if condition is true then check the following
      if( b == 200 ) {
         // if condition is true then print the following
         cout << "Value of a is 100 and b is 200" << endl;
      }
   }
   cout << "Exact value of a is : " << a << endl;
   cout << "Exact value of b is : " << b << endl;
   return 0;
}
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

5. Nested switch statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Syntax

The syntax for a **nested switch** statement is as follows −

```
switch(ch1) {
   case 'A':
      cout << "This A is part of outer switch";
      switch(ch2) {
         case 'A':
            cout << "This A is part of inner switch";
            break;
         case 'B': // ...
      }
      break;
   case 'B': // ...
}
```

Example

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

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

   switch(a) {
      case 100:
         cout << "This is part of outer switch" << endl;
         switch(b) {
            case 200:
               cout << "This is part of inner switch" << endl;
         }
   }
   cout << "Exact value of a is : " << a << endl;
   cout << "Exact value of b is : " << b << endl;

   return 0;
}
```

This would produce the following result −

```
This is part of outer switch
```

```
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

Scope Resolution Operator

The **:: (scope resolution) operator** is used to get hidden names due to variable scopes so that you can still use them. The scope resolution operator can be used as both unary and binary. You can use the unary scope operator if a namespace scope or global scope name is hidden by a particular declaration of an equivalent name during a block or class. For example, if you have a global variable of name my_var and a local variable of name my_var, to access global my_var, you'll need to use the scope resolution operator.

example

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

int my_var = 0;
int main(void) {
   int my_var = 0;
   ::my_var = 1;  // set global my_var to 1
   my_var = 2;    // set local my_var to 2
   cout << ::my_var << ", " << my_var;
   return 0;
}
```

Output

This will give the output −

```
1, 2
```

The declaration of **my_var** declared in the main function hides the integer named **my_var declared in global namespace scope**. The statement **::my_var = 1** accesses the variable named **my_var** declared in **global namespace scope**.
You can also use the scope resolution operator to use class names or class member names. If a class member name is hidden, you can use it by prefixing it with its class name and the class scope operator. For example,

Example

```
#include <iostream>
using namespace std;
class X {
   public:
   static int count;
};
int X::count = 10;  // define static data member

int main () {
   int X = 0;   // hides class type X
   cout << X::count << endl;   // use static member of class X
}
```

Output

```
10
```

Memory Allocation Operators

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

New and Delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows −

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how 'new' and 'delete' work −

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of pvalue : 29495
```

Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;          // Pointer initialized with null
pvalue  = new char[20];        // Request memory for the variable
```

To remove the array that we have just created the statement would look like this −

```
delete [] pvalue;              // Delete array pointed to by pvalue
```

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

```
double** pvalue  = NULL;       // Pointer initialized with null
pvalue  = new double [3][4];   // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

```
delete [] pvalue;             // Delete array pointed to by pvalue
```

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

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

```
class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Arrays

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

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

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

Declaring Arrays

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

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

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

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

Initializing Arrays

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

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

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

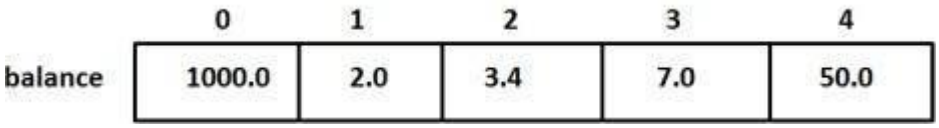
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

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

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

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

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



Accessing Array Elements

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

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

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

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

#include <iomanip>
using std::setw;

int main () {

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

   // initialize elements of array n to 0
```

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

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

   return 0;
}
```

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

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

Arrays in C++

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

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

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**.

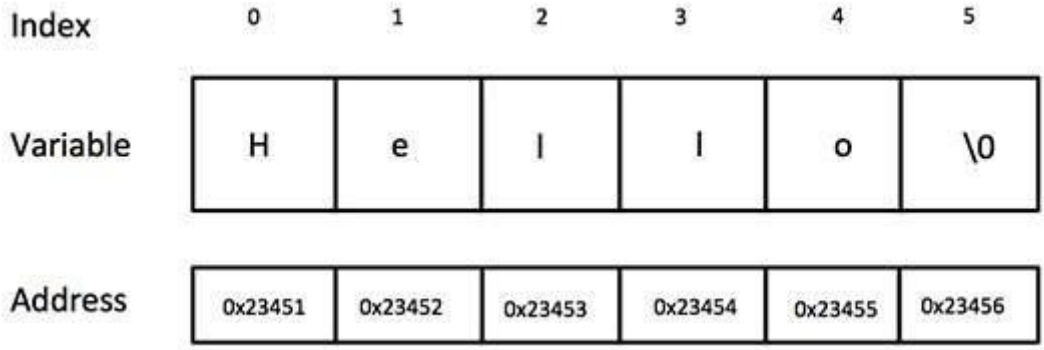
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
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
```

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
```