<u>Department of Computer Science & Engineering</u>

SLIET Longowal

Course Material

of

Object Oriented Programming (CS-221/PCCS-202)

for

Integrated Certificate & Diploma in Computer Science (ICD-CS) 4th Sem

Submitted by

Er. Rahul Gautam (A.P., CSE)

Title of the course	: **Object Oriented Programming**		
Subject Code	: **CS-221**		
Weekly load	: 6 Hrs	LTP	2-0-4
Credit	: 4 (Lecture 2, Practical 2)		

Course Outcomes: At the end of the course, the student will be able to:

CO1	Apply object-oriented approach to design the programs.
CO2	Understand reusability of code using inheritance.
CO3	Analyze polymorphic and virtual behaviour of functions.
CO4	Use stream classes in file-handling.

CO/PO Mapping : (Strong(S)/Medium(M)/Weak(W) indicates strength of correlation)

COs	Programme Outcomes (POs)									
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10
CO1		S	S	M						S
CO2		S	M	W						S
CO3		S	S	W						S
CO4		S	S	M						S

Theory

Unit	**Main Topics**	**Course outlines**	**Lecture(s)**
Unit-1	1. Introduction	Object-oriented programming, characteristics of object-oriented languages, C++ Programming Basics: Basic program construction, Pre-processor directives, variables, Operators, Library functions, manipulators.	04
	2. Decision-making	Relational operators: loops; decisions; logical operators; other control Statements	04
	3. Structures and Functions	Structure enumerated data types; functions; passing arguments to functions and returning values from functions, unions.	03
	4. Classes and Objects	Creation, accessing class members, Private Vs Public, Constructor and Destructor Objects.	03
Unit-2	5. Member Functions	Method definition, Inline functions implementation, Constant member functions Friend Functions and Friend Classes, Static functions Overloading Member Functions, Need of operator overloading, operator overloading	05
	6.Inheritance	Definition of inheritance, protected data, private data, public data, inheriting constructors and destructors, constructor for virtual base classes, constructors and destructors of derived classes, size of a derived class, order of invocation, types of inheritance, single inheritance, hierarchical inheritance, multiple inheritance, hybrid inheritance, multilevel inheritance.	05

	7. Polymorphism and Virtual Functions	Importance of virtual function, function call binding, implementing late binding, need for virtual functions, abstract base classes and pure virtual functions, virtual destructors	04
	8. File and Streams	File and Streams components of a file, different operation of the file, creation of file streams, stream classes, header files, updating of file, opening and closing a file.	04

Total=32

Recommended Books:

1. SB Lippman and J Lajoie,C++ Primer, Addison Wesley ,New Delhi
2. KR Venugopal , Mastering C++ , TMH Publishing
3. E. Balaguruswamy, Object Oriented Programming in C++, TMH Publishing Co. Ltd, New Delhi.
4. Robert Lafore, C++, Galgotia Publications Pvt. Ltd., Daryaganj, New Delhi.

Object Oriented Programming

UNIT-1

1. INTRODUCTION

<u>Object Oriented Programming & their characteristics</u>

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use 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.

OOPs Concepts:

- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

1. Class:

A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. 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 brands 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.

2. Object:

It is a basic unit of Object-Oriented Programming and represents the real-life entities. 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. An object has an identity, state, and behavior. 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.

For example "Dog" is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

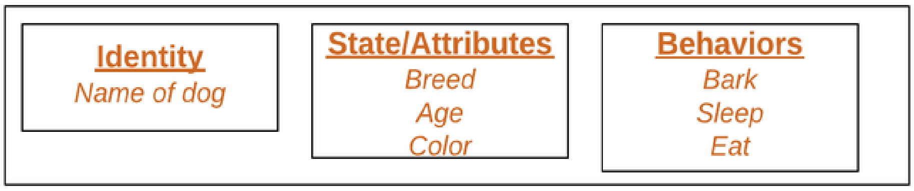
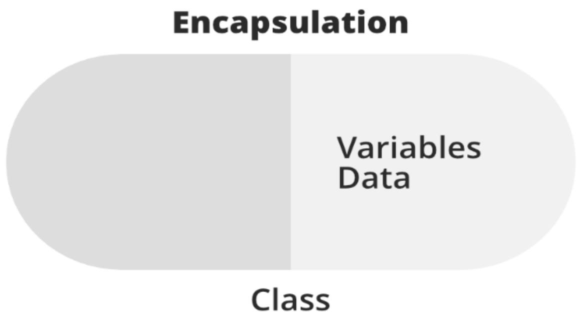


Fig 1.1 Example of an Object

3. Data Abstraction:
Data abstraction is one of the most essential and important features of object-oriented programming. 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 the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

4. Encapsulation:
Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.



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”.

5. Inheritance:
Inheritance is an important pillar of OOP(Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

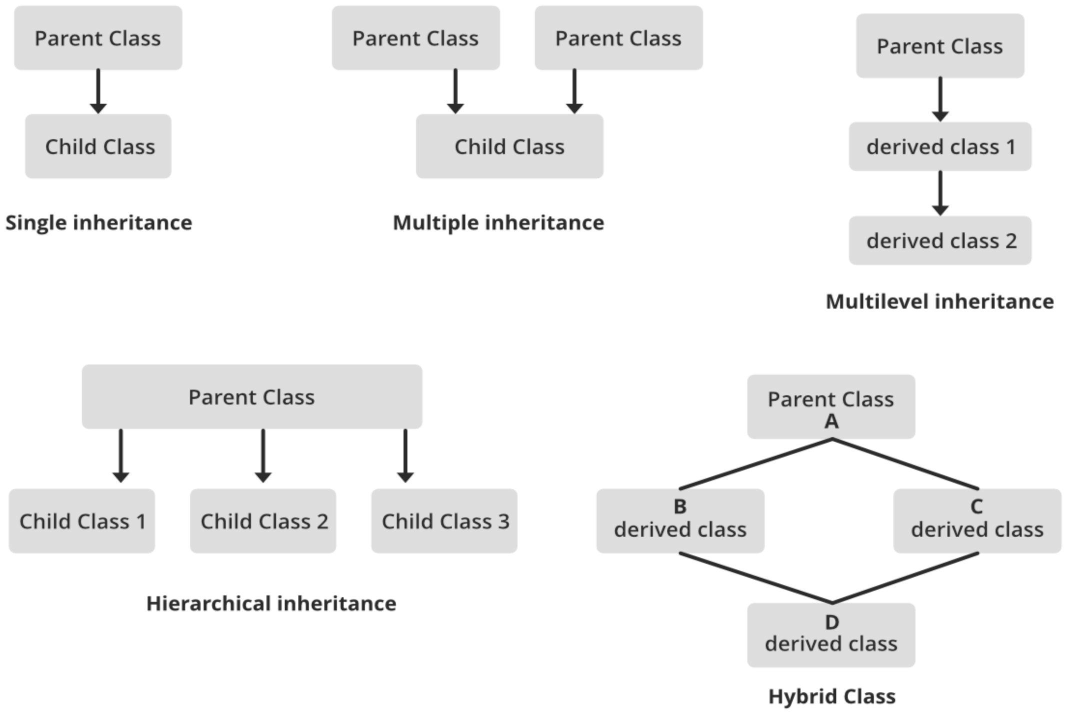


Fig 1.2 Types of Inheritance

6. 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. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

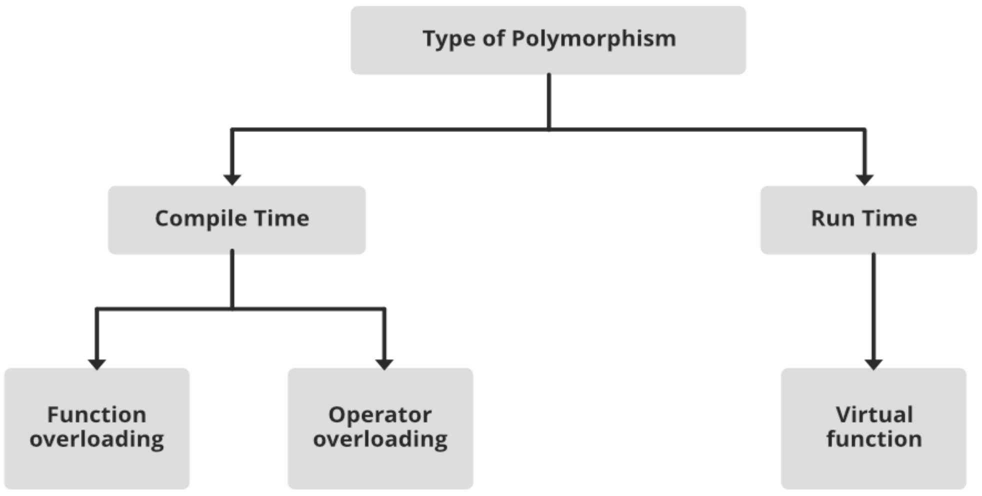


Fig 1.3 Types of Polymorphism

7. Dynamic Binding:

In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B. Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used. This feature is known as subtype polymorphism.

8. Message Passing:
It is a form of communication used in object-oriented programming as well as parallel programming. 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.

Why do we need object-oriented programming

- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.

C++ Programming Basics:

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.

1. C++ is a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.
2. One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high-level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications.
3. C++ has a large, active community of developers and users, and a wealth of resources and tools available for learning and using the language. Some of the key features of C++ include:
4. Object-Oriented Programming: C++ supports object-oriented programming, allowing developers to create classes and objects and to define methods and properties for these objects.
5. Templates: C++ templates allow developers to write generic code that can work with any data type, making it easier to write reusable and flexible code.
6. Standard Template Library (STL): The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.

7. Exception Handling: C++ provides robust exception handling capabilities, making it easier to write code that can handle errors and unexpected situations.

Overall, C++ is a powerful and versatile programming language that is widely used for a range of applications and is well-suited for both low-level system programming and high-level application development.

Basic Program construction:

Here are some simple C++ code examples to help you understand the language:

1.Hello World:

```
#include <iostream>
int main() {
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

Output

```
Hello, World!
```

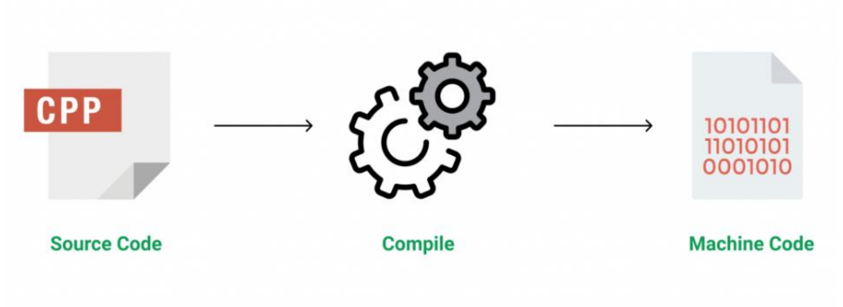


Fig 1.4 Execution sequence of a program

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.). The basic syntax and code structure of both C and C++ are the same. 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.

Here are some key points to keep in mind while working with C++:

1. Object-Oriented Programming: C++ is an object-oriented programming language, which means that it allows you to define classes and objects to model real-world entities and their behavior.
2. Strong Type System: C++ has a strong type system, which means that variables have a specific type and that type must be respected in all operations performed on that variable.
3. Low-level Access: C++ provides low-level access to system resources, which makes it a suitable choice for system programming and writing efficient code.
4. Standard Template Library (STL): The STL provides a large set of pre-written algorithms and data structures that can be used to simplify your code and make it more efficient.
5. Cross-platform Compatibility: C++ can be compiled and run on multiple platforms, including Windows, MacOS, and Linux, making it a versatile language for developing cross-platform applications.
6. Performance: C++ is a compiled language, which means that code is transformed into machine code before it is executed. This can result in faster execution times and improved performance compared to interpreted languages like Python.
7. Memory Management: C++ requires manual memory management, which can lead to errors if not done correctly. However, this also provides more control over the program's memory usage and can result in more efficient memory usage.
8. Syntax: C++ has a complex syntax that can be difficult to learn, especially for beginners. However, with practice and experience, it becomes easier to understand and work with.

These are some of the key points to keep in mind when working with C++. By understanding these concepts, you can make informed decisions and write effective code in this language.

Applications of C++:

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
- Browsers *(Chrome & Firefox)*
- Graphics & Game engines *(Photoshop, Blender, Unreal-Engine)*
- Database Engines *(MySQL, MongoDB, Redis etc.)*
- Cloud/Distributed Systems

Advantages of C++:

1. Performance: C++ is a compiled language, which means that its code is compiled into machine-readable code, making it one of the fastest programming languages.
2. Object-Oriented Programming: C++ supports object-oriented programming, which makes it easier to write and maintain large, complex applications.
3. Standard Template Library (STL): The STL provides a wide range of algorithms and data structures for working with data, making it easier to write efficient and effective code.
4. Machine Independent: C++ is not tied to any hardware or processor. If the compiler compiles the program in the system, it will be able to run no matter what the hardware is.
5. Large Community: C++ has a large, active community of developers and users, providing a wealth of resources and support for learning and using the language.

Disadvantages of C++:

1. Steep Learning Curve: C++ can be challenging to learn, especially for beginners, due to its complexity and the number of concepts that need to be understood.
2. Verbose Syntax: C++ has a verbose syntax, which can make code longer and more difficult to read and maintain.
3. Error-Prone: C++ provides low-level access to system resources, which can lead to subtle errors that are difficult to detect and fix.

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)

Setting up C++ Development Environment

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc. Before we start programming with C++. We will need an environment to be set up on our local computer to compile and run our C++ programs successfully. If you do not want to set up a local environment, you can also use online IDEs for compiling your program.

Using Online IDE

IDE stands for an integrated development environment. IDE is a software application that provides facilities to a computer programmer for developing software. There are many online IDEs available that you can use to compile and run your programs easily without setting up a local development environment.

```
// Using online ide of C++
#include <iostream>
using namespace std;

int main()
{
    cout << "Learning C++";
    return 0;
}
```

Output

```
Learning C++
```

Setting up a Local Environment:
For setting up a C++ Integrated Development Environment (IDE) on your local machine you need to install two important software:

1. C++ Compiler
2. Text Editor

1. C++ Compiler

Once you have installed the text editor and saved your program in a file with the '.cpp' extension, you will need a C++ compiler to compile this file. A compiler is a computer program that converts high-level language into machine-understandable low-level language. In other words, we can say that it converts the source code written in a programming language into another computer language that the computer understands. For compiling a C++ program we will need a C++ compiler that will convert the source code written in C++ into machine codes. Below are the details about setting up compilers on different platforms.

Installing GNU GCC on Linux
We will install the GNU GCC compiler on Linux. To install and work with the GCC compiler on your Linux machine, proceed according to the below steps:

A. You have to first run the below two commands from your Linux terminal window:

```
sudo apt-get update

sudo apt-get install gcc

sudo apt-get install g++
```

B. This command will install the GCC compiler on your system. You may also run the below command:

```
sudo apt-get install build-essential
```

This command will install all the libraries which are required to compile and run a C++ program.

C. After completing the above step, you should check whether the GCC compiler is installed in your system correctly or not. To do this you have to run the below-given command from the Linux terminal:

```
g++ --version
```

D. If you have completed the above two steps without any errors, then your Linux environment is set up and ready to be used to compile C++ programs. In further steps, we will learn how to compile and run a C++ program on Linux using the GCC compiler.

E. Write your program in a text file and save it with any file name and.CPP extension. We have written a program to display “Hello World” and saved it in a file with the filename “helloworld.cpp” on the desktop.

F. Now you have to open the Linux terminal and move to the directory where you have saved your file. Then you have to run the below command to compile your file:

```
g++ filename.cpp -o any-name
```

G. *filename.cpp* is the name of your source code file. In our case, the name is “helloworld.cpp” and *any-name* can be any name of your choice. This name will be assigned to the executable file which is created by the compiler after compilation. In our case, we choose *any-name* to be “hello”.
We will run the above command as:

```
g++ helloworld.cpp -o hello
```

H. After executing the above command, you will see a new file is created automatically in the same directory where you have saved the source file, and the name of this file is the name you chose as *any-name*. Now to run your program you have to run the below command:

```
./hello
```

I. This command will run your program in the terminal windows.

2. Text Editor

Text Editors are the type of programs used to edit or write texts. We will use text editors to type our C++ programs. The normal extension of a text file is (.txt) but a text file containing a C++ program should be saved with a ‘.cpp’ or ‘.c’ extension. Files ending with the extension ‘.CPP’ and ‘.C’ are called source code files and they are supposed to contain source code written in C++ programming language. These extension helps the compiler to identify that the file contains a C++ program.
Before beginning programming with C++, one must have a text editor installed to write programs. Follow the below instructions to install popular code editors like VS Code and Code::Block on different Operating Systems like windows, Mac OS, etc.

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 most straightforward programs you will learn. It is the basic program that is used to demonstrate how the coding process works. All you have to do is display the message “Hello World” on the output screen.

Below is the C++ program to print “Hello World” on the console screen.

```
// Header file for input output functions
#include <iostream>
using namespace std;
```

```
// main() function: where the execution of
// C++ program begins
int main() {

   // This statement prints "Hello World"
   cout << "Hello World";

   return 0;
}
```
Output

```
Hello World
```

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

// 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.

#include <iostream>

The **#include** is a preprocessor directive tells the compiler to include the content of a file in the source code. For example, **#include<iostream>** tells the compiler to include the input-output library which contains all C++'s input/output library functions.

using namespace std

This is used to import the entity of the **std** namespace into the current namespace of the program. It is basically the space where all the inbuilt features of C++ are declared. For example, std::cout.

int main() { }

The *main()* function is the entry point of every C++ program, no matter where the function is located in the program. The opening braces '{'indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function.

cout<<"Hello World";

The *cout* is a tool (object) that is used to display output on the console screen. Everything followed by the character << in double quotes" " is displayed on the output screen. The semi-colon character at the end of the statement is used to indicate that the statement is ending there.

return 0;

This statement is used to return a value from a function and indicates the finishing of a function. Here, it is used to sent the signal of successful execution of the main function.

C++ Variables

In C++, **variable** is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be accessed or changed during program execution.

Creating a Variable:

Creating a variable and giving it a name is called **variable definition** (sometimes called **variable declaration**). The syntax of variable definition is:

```
type name;
```

where, **type** is the type of data that a variable can store, and **name** is the name assigned to the variable. Multiple variables of the same type can be defined as:

```
type name1, name2, name3 ....;
```

The data type of a variable is selected from the list of data types supported by C++.
Example:
To store number without decimal point, we use integer data type.

```
int num;
```

Here, **int** is the keyword used to tell the compiler that the variable with name **num** will store integer values.
Initializing:
A variable that is just defined may not contain some valid value. We have to initialize it to some valid initial value. It is done by using an assignment operator = as shown:

```
type name;
name = value;
```

Definition and initialization can also be done in a single step as shown:

```
type name = value;
```

where the **value** should be of the same type as variable.
Example:

```
int num = 100;
```

The integer variable **num** is initialized with the value 100. Values are generally the literals of the same type.
Accessing:
The main objective of a variable is to store the data so that it can be retrieved later. It can be done by simply using its assigned name.
Example:

```
 {...}
   // Creating a single character variable
   char c = 'a';

   // Accessing and printing above variable
   cout << c;

 {...}
```

Output

```
a
```

Updating:
The value stored in the variables can be changed any number of times by simply assigning a new value using **= assignment operator.**
Example:

```
 {...}
   int num = 24;
   cout << num << endl;

   // Assigning new value
   num = 888;
   cout << num << endl;
 {...}
```

Output

```
24
```

```
888
```

Rules For Naming Variable

The names given to a variable are called identifiers. There are some rules for creating these identifiers (names):

- A name can only contain **letters** (A-Z or a-z), **digits** (0-9), and **underscores (_)**.
- It should start with a **letter or an underscore only.**
- It is case sensitive.
- The name of the variable should not contain any whitespace and special characters (i.e. #, $, %, *, etc).
- We cannot use C++ keywords (e.g. float, double, class) as a variable name.

How are variables used?

Variables are the names given to the memory location which stores some value. These names can be used in any place where the value it stores can be used. **For example,** we assign values of the same type to variables. But instead of these values, we can also use variables that store these values.

```
{...}
   int num1 = 10, num2;

   // Assigning num1's value to num2
   num2 = num1;
   cout << num1 << " " << num2;

{...}
```

Output

```
10 10
```

Addition of two integers can be done in C++ using + operator as shown:

```
{...}
   cout << 10 + 20;

{...}
```

Output

```
30
```

We can do the above operation using the variables that store these two values.

```
{...}
   int num1 = 10, num2 = 20;
   cout << num1 + num2;

{...}
```

Output

```
30
```

Constant Variables

In C++, a constant variable is one whose value cannot be changed after it is initialized. This is done using the const keyword.

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

int main() {
   const int num = 10;
   cout << num;
   return 0;
}
```
Scope of Variables is the region inside the program where the variable can be referred to by using its name. Basically, it is the part of the program where the variable exists. Proper understanding of this concept requires the understanding of other concepts such as functions, blocks, etc.
Memory Management of Variables:
When we create or declare a variable, a fixed-size memory block is assigned to the variable, and its initial value is a garbage value. Initialization assigns a meaningful value using the assignment operator. Variables essentially manipulate specific memory locations, and their stored data is accessed via their names.

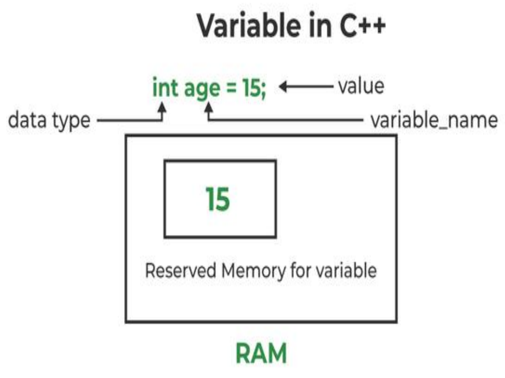


Fig 1.5 storing a variable

Moreover, different variables may be stored in different section of memory according to its storage class.

C++ Data Types:

In C++, data types are classified into the following types:

Table 1.1 Data Types

S. No.	**Type**	**Description**	**Data Types**
1	Basic Data Types	Built-in or primitive data types that are used to store simple values.	**Int, float, double, char, bool, void**
2	Derived Data Types	Data types derived from basic types.	**Array, pointer, reference, function**

S. No.	Type	Description	Data Types
3	User defined Data Types	Custom data types created by the programmer according to their need.	**Class, struct, union, typedef, using**

Let’s see how to use some primitive data types in C++ program.

1. Character Data Type (char)

The character data type is used to store a single character. The keyword used to define a character is **char**. Its size is 1 byte and it stores characters enclosed in single quotes (‘ ‘). It can generally store upto 256 characters according to their ASCII codes.

Syntax

***char** name;*

where ***name*** is the identifier assigned to the variable.

Example

```
#include <iostream>
using namespace std;
int main() {
     // Character variable
     char c = ‘A’;
    cout << c;
    return 0;
}
```

Output

```
A
```

1. Integer Data Type (int)

Integer data type denotes that the given variable can store the integer numbers. The keyword used to define integers is **int.** Its size is **4-bytes** (for 64-bit compiler) and can store numbers for binary, octal, decimal and hexadecimal base systems in the range from **-2,147,483,648 to 2,147,483,647.**

Syntax

***int** name;*

where, ***name*** is the identifier assigned to the variable.

Example

```
#include <iostream>
using namespace std;
int main() {
     // Creating an integer variable
    int x = 25;
    cout << x << endl;
     // Using hexadecimal base value
     x = 0x15;
     cout << x;
    return 0;
}
```

Output

```
25

21
```

1. Boolean Data Type (bool)

The Boolean data type is used to store logical values: **true(1)** or **false(0)**. The keyword used to define a 17oolean variable is **bool**. Its size is 1 byte.

Syntax

bool *name;*

where ***name*** is the identifier assigned to the variable.

Example

```
#include <iostream>
using namespace std;
int main() {
     // Creating a 17oolean variable
   bool isTrue = true;
   cout << isTrue;
   return 0;
}
```

Output

```
1
```

1. Floating Point Data Type (float)

Floating-point data type is used to store numbers with decimal points. The keyword used to define floating-point numbers is **float**. Its size is 4 bytes (on 64-bit systems) and can store values in the range from **1.2E-38 to 3.4e+38.**

Syntax

float *name;*

where, ***name*** is the identifier assigned to the variable.

```
#include <iostream>
using namespace std;
int main() {
     // Floating point variable with a decimal value
     float f = 36.5;
     cout << f;
   return 0;
}
```

Output

```
36.5
```

5. Double Data Type (double)

The **double data type** is used to store decimal numbers with higher precision. The keyword used to define double-precision floating-point numbers is **double**. Its size is 8 bytes (on 64-bit systems) and can store the values in the range from **1.7e-308 to 1.7e+308**

Syntax

double *name;*

where, ***name*** is the identifier assigned to the variable.

Example

```
#include <iostream>
using namespace std;
int main() {
      // double precision floating point variable
      double pi = 3.1415926535;
      cout << pi;
    return 0;
}
```

Output

```
3.14159
```

6. Void Data Type (void)

The **void data type** represents the absence of value. We cannot create a variable of void type. It is used for pointer and functions that do not return any value using the keyword **void**.

Syntax

void *functionName();*

Example

```
#include <iostream>
using namespace std;
// Function with void return type
void hello() {
    cout << “Hello, World!” << endl;
}
int main() {
      hello();
    return 0;
}
```

Output

```
Hello, World!
```

Operators in C++:

In C++, an **operator** is a symbol that operates on a value to perform specific mathematical or logical computations on given values. They are the foundation of any programming language.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int a = 10 + 20;
    cout << a;
    return 0;
}
```

Output

```
30
```

Explanation: Here, '**+**' is an **addition operator** and does the addition of 10 and 20 operands and return value 30 as a result.
In C++, operators are classified into 6 types on the basis of type of operation they perform:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary or Conditional Operators

1. Arithmetic Operators

Arithmetic Operators are used to perform arithmetic or mathematical operations on the operands. For example, '+' is used for addition.

Table 1.2 Arithmetic Operators

Name	**Symbol**	**Description**
Addition	+	Adds two operands.
Subtraction	–	Subtracts second operand from the first.
Multiplication	*	Multiplies two operands.
Division	/	Divides first operand by the second operand.
Modulo Operation	%	Returns the remainder an integer division.
Increment	++	Increase the value of operand by 1.
Decrement	--	Decrease the value of operand by 1.

Example:
```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 8, b = 3;
   // Addition
   cout << "a + b = " << (a + b) << endl;
   // Subtraction
   cout << "a – b = " << (a – b) << endl;
   // Multiplication
   cout << "a * b = " << (a * b) << endl;
   // Division
   cout << "a / b = " << (a / b) << endl;
   // Modulo
   cout << "a % b = " << (a % b) << endl;
   // Increament
   cout << "++a = " << ++a << endl;
   // Decrement
   cout << "–b = " << --b;
   return 0;
}
```

Output

```
a + b = 11

a – b = 5

a * b = 24

a / b = 2

a % b = 2

++a = 9

--b = 2
```

Important Points:

- The Modulo operator (%) operator should only be used with integers. Other operators can also be used with floating point values.
- **++a** and **a++**, both are increment operators, however, both are slightly different. In **++a**, the value of the variable is incremented first and then It is used in the program. In **a++**, the value of the variable is assigned first and then It is incremented. Similarly happens for the decrement operator.

You may have noticed that some operator works on two operands while other work on one. On the basis of this operators are also classified as:

- **Unary**: Works on single operand.
- **Binary**: Works on two operands.
- **Ternary**: Works on three operands.

2. Relational Operators

Relational Operators are used for the comparison of the values of two operands. For example, '>' check right operand is greater.

Table 1.3 Relational Operators

Name	**Symbol**	**Description**
Is Equal To	==	Checks both operands are equal
Greater Than	>	Checks first operand is greater than the second operand

Name	Symbol	Description
Greater Than or Equal To	>=	Checks first operand is greater than equal to the second operand
Less Than	<	Checks first operand is lesser than the second operand
Less Than or Equal To	<=	Checks first operand is lesser than equal to the second operand
Not Equal To	!=	Checks both operands are not equal

Example

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 6, b = 4;

   // Equal operator
   cout << "a == b is " << (a == b) << endl;
   // Greater than operator
   cout << "a > b is " << (a > b) << endl;
   // Greater than Equal to operator
   cout << "a >= b is " << (a >= b) << endl;
   //  Lesser than operator
   cout << "a < b is " << (a < b) << endl;
   // Lesser than Equal to operator
   cout << "a <= b is " << (a <= b) << endl;
   // Not equal to operator
   cout << "a != b is " << (a != b);
   return 0;
}
```

Output

```
a == b is 0

a > b is 1

a >= b is 1

a < b is 0

a <= b is 0

a != b is 1
```

***Note:** 0 denotes **false** and 1 denotes **true**.*

3. Logical Operators

Logical Operators are used to combine two or more conditions or constraints or to complement the evaluation of the original condition in consideration. The result returns a Boolean value, i.e., true or false.

Table 1.4 Logical Operators

Name	**Symbol**	**Description**
Logical AND	**&&**	Returns true only if all the operands are true or non-zero.
Logical OR	||	Returns true if either of the operands is true or non-zero.
Logical NOT	**!**	Returns true if the operand is false or zero.

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 6, b = 4;
   // Logical AND operator
   cout << "a && b is " << (a && b) << endl;
   // Logical OR operator
   cout << "a || b is " << (a || b) << endl;
   // Logical NOT operator
   cout << "!b is " << (!b);
   return 0;
}
```

Output

```
a && b is 1

a || b is 1

!b is 0
```

4. Bitwise Operators

Bitwise Operators are works on bit-level. So, compiler first converted to bit-level and then the calculation is performed on the operands.

Table 1.5 Bitwise Operators

Name	**Symbol**	**Description**
Binary AND	**&**	Copies a bit to the evaluated result if it exists in both operands

Name	Symbol	Description
Binary OR	|	Copies a bit to the evaluated result if it exists in any of the operand
Binary XOR	^	Copies the bit to the evaluated result if it is present in either of the operands but not both
Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.
Right Shift	>>	Shifts the value to right by the number of bits specified by the right operand.
One's Complement	~	Changes binary digits 1 to 0 and 0 to 1

Note: *Only* ***char*** *and* ***int*** *data types can be used with Bitwise Operators.*

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 6, b = 4;
   // Binary AND operator
   cout << "a & b is " << (a & b) << endl;
   // Binary OR operator
   cout << "a | b is " << (a | b) << endl;
   // Binary XOR operator
   cout << "a ^ b is " << (a ^ b) << endl;
   // Left Shift operator
   cout << "a<<1 is " << (a << 1) << endl;
   // Right Shift operator
   cout << "a>>1 is " << (a >> 1) << endl;
   // One's Complement operator
   cout << "~(a) is " << ~(a);
   return 0;
}
```

Output

```
a & b is 4

a | b is 6
```

```
a ^ b is 2
a<<1 is 12
a>>1 is 3
~(a) is -7
```

6. Assignment Operators

Assignment Operators are used to assign value to a variable. We assign the value of right operand into left operand according to which assignment operator we use.

Table 1.6 Assignment Operators

Name	**Symbol**	**Description**
Assignment	=	Assigns the value on the right to the variable on the left.
Add and Assignment	+=	First add right operand value into left operand then assign that value into left operand.
Subtract and Assignment	-=	First subtract right operand value into left operand then assign that value into left operand.
Multiply and Assignment	*=	First multiply right operand value into left operand then assign that value into left operand.
Divide and Assignment	/=	First divide right operand value into left operand then assign that value into left operand.

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int a = 6, b = 4;
   // Assignment Operator.
   cout << "a = " << a << endl;
   //  Add and Assignment Operator.
   cout << "a += b is " << (a += b) << endl;
   // Subtract and Assignment Operator.
   cout << "a -= b is " << (a -= b) << endl;
   //  Multiply and Assignment Operator.
   cout << "a *= b is " << (a *= b) << endl;
   //  Divide and Assignment Operator.
   cout << "a /= b is " << (a /= b);
   return 0;
}
```

Output

```
a = 6

a += b is 10

a -= b is 6

a *= b is 24

a /= b is 6
```

7. Ternary or Conditional Operators

Ternary or conditional operators returns the value, based on the condition.
Expression1 ? Expression2 : Expression3
The ternary operator **?** determines the answer on the basis of the evaluation of **Expression1**. If it is **true**, then **Expression2** gets evaluated and is used as the answer for the expression. If **Expression1** is **false**, then **Expression3** gets evaluated and is used as the answer for the expression.
This operator takes **three operands**, therefore it is known as a Ternary Operator.
Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int a = 3, b = 4;
    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result;
    return 0;
}
```

Output

```
The greatest number is 4
```

Basic Input / Output in C++:

In C++, input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device (display screen) then this process is called output.

All of these streams are defined inside the **<iostream>** header file which contains all the standard input and output tools of C++. The two instances **cout** and **cin** of iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++.

Standard Output Stream – cout:
The C++ cout is the instance of the **ostream** class used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the **insertion operator (<<)**.
Syntax
cout << *value/variable;*
Standard Input Stream – cin:
The C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard. The **extraction operator (>>)** is used along with the object **cin** for extracting the data from the input stream and store it in some variable in the program.
Syntax
cin >> *variable;*
For example, if we want to ask user for his/her age, then we can use cin as shown:

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

int main() {
   int age;
     // Output a label
   cout << "Enter your age:";
     // Taking input from user and store
     // it in variable
   cin >> age;
     // Output the entered age
   cout << "Age entered: " << age;
   return 0;
}
```

Input

```
Enter your age: 18 (18 entered by the user)
```

Output

```
Your age is: 18
```

Manipulators in C++:

Manipulators are helping functions that can modify the input or output stream. They can be included in the I/O statement to alter the format parameters of a stream. They are defined inside <iomanip> and some are also defined inside <iostream> header file.
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 classified on the basis type of entity they manipulate:

- Output Stream Manipulators
- Input Stream MAnipulators
- Boolean Manipulators
- Alignment and Sign Manipulators
- Base Manipulators

1. Output Stream Manipulators

Output stream manipulators are used to control and format the output stream, such as setting the width, precision, or alignment of printed data. They allow for a better presentation of output.
Following table lists some common output stream manipulators:

Table 1.7 Output stream manipulators

Manipulator	**Description**	**Header File**
endl	Inserts a newline and flushes the output stream.	iostream
flush	Flushes the output stream manually.	iostream
setw(x)	Sets the width of the next output field to x.	iomanip
setprecision(x)	Sets the precision for floating-point numbers to x.	iomanip
fixed	Displays numbers in fixed-point notation.	iomanip
scientific	Displays numbers in scientific notation.	iomanip
showpoint	Forces the display of the decimal point.	iomanip
noshowpoint	Hides the decimal point unless necessary.	iomanip

Example

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

int main() {
   // Output a new line and flush the stream
   cout << "Hello" << endl;
   // Set width to 10 for the next output
   cout << setw(10) << 42 << endl;
   // Set precision to 3 for floating-point numbers
   cout << setprecision(3) << 3.14159 << endl;
   // Use fixed-point notation
   cout << fixed << 3.14159 << endl;
   // Use scientific notation
   cout << scientific << 3.14159 << endl;
```

```
    // Show the decimal point even for whole numbers
    cout << showpoint << 42.0;
    return 0;
}
```

Output

```
Hello

       42

3.14

3.142

3.142e+00

4.200e+01
```

2. Input Stream Manipulators

Input stream manipulators are used to modify the behaviour of the input stream. They help in processing input efficiently, such as skipping unnecessary whitespaces with ws.

Following table lists some common input stream manipulators:

Table 1.8 Input stream manipulators

Manipulator	**Description**	**Header File**
ws	Skips leading whitespaces in the input stream.	iostream
noskipws	Disables skipping of leading whitespaces.	iostream

Example

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

int main() {
    char c1, c2;
    // Input skips whitespace by default
    cin >> c1;
    // Input the next character without skipping whitespace
    cin >> noskipws >> c2;
    cout << "c1: " << c1 << ", c2: " << c2;
    return 0;
}
```

Input

```
   s   x
```

Output

```
c1: s, c2:
```

3. Boolean Manipulators

Boolean manipulators are used to format boolean values in output. They allow displaying boolean values as true or false or as 1 and 0, depending on the requirement.

Following table lists some common boolean manipulators:

Table 1.9 Boolean manipulators

Manipulator	Description	Header File
boolalpha	Displays true or false for boolean values.	iostream
noboolalpha	Displays 1 or 0 for boolean values.	iostream

Example

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

int main() {
   bool value = true;
   // Display boolean as true/false
   cout << boolalpha << value << endl;
   // Display boolean as 1/0
   cout << noboolalpha << value;
   return 0;
}
```

Output

```
true

1
```

4. Alignment and Sign Manipulators

These manipulators control how text and numbers are aligned or how their signs are displayed in the output.

Following table lists some common alignment and sign manipulators:

Table 1.10 Alignment and Sign manipulators

Manipulator	Description	Header File
left	Aligns output to the left.	Iomanip
right	Aligns output to the right.	Iomanip
internal	Aligns signs and base prefixes to the left.	Iomanip

Manipulator	Description	Header File
showpos	Displays a + sign for positive numbers.	Iostream
noshowpos	Hides the + sign for positive numbers.	Iostream

Example

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

int main() {
   int n = 42;
   // Align output to the left
   cout << left << setw(10) << n << endl;
   // Align output to the right
   cout << right << setw(10) << n << endl;
   // Show positive sign for numbers
   cout << showpos << n << endl;
   // Don't show positive sign for numbers
   cout << noshowpos << n;
   return 0;
}
```

Output

```
42
        42
+42
42
```

4. Base Manipulators

Base manipulators are used to format numbers in different bases, such as decimal, hexadecimal, or octal. They help in representing numbers in a way suited to specific applications.

Following table lists some common base manipulators:

Table 1.11 Base manipulators

Manipulator	Description	Header File
hex	Formats output in hexadecimal base.	iostream
dec	Formats output in decimal base.	iostream

Manipulator	Description	Header File
oct	Formats output in octal base.	iostream

Example

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

int main() {
    int n = 42;
    // Output in hexadecimal base
    cout << hex << n << endl;
    // Output in decimal base
    cout << dec << n << endl;
    // Output in octal base
    cout << oct << n;
    return 0;
}
```

Output

```
2a

42

52
```

2. DECISION MAKING

Decision Making in C/C++ (if , if..else, Nested if, if-else-if):

The **conditional statements** (also known as decision control structures) such as if, if else, switch, etc. are used for decision-making purposes in C/C++ programs.

They are also known as Decision-Making Statements and are used to evaluate one or more conditions and make the decision whether to execute a set of statements or not. These decision-making statements in programming languages decide the direction of the flow of program execution.
Need of Conditional Statements:
There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C/C++ if x occurs then execute y else execute z. There can also be multiple conditions like in C/C++ if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C/C++ else-if is one of the many ways of importing multiple conditions.
Types of Conditional Statements in C/C++:

1. **if Statement**
2. **if-else Statement**
3. **Nested if Statement**
4. **if-else-if Ladder**
5. **switch Statement**
6. **Jump Statements:**
    - **break**
    - **continue**
    - **goto**

1. if statement
The if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e., if a certain condition is true then a block of statements is executed otherwise not.

Syntax of if Statement

```
if(condition)
{
  //Statements to execute if
  // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C/C++ if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

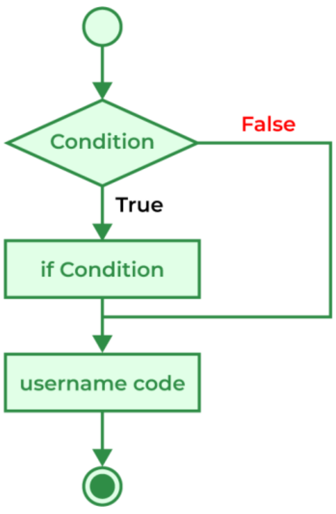


Figure 2.1: Flow Diagram of if Statement

Example of if statement

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
   int i = 10;
   if (i > 15) {
      printf("10 is greater than 15");
   }
   printf("I am Not in if");
}
```

Output

```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

2. if-else statement

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else when the condition is false? Here comes the C/C++ *else* statement. We can use the *else* statement with the *if* statement to execute a block of code when the condition is false. The if-else consists of two blocks, one for false expression and one for true expression.

Syntax of if else in C

```
if(condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
```

```
    // condition is false
}
```

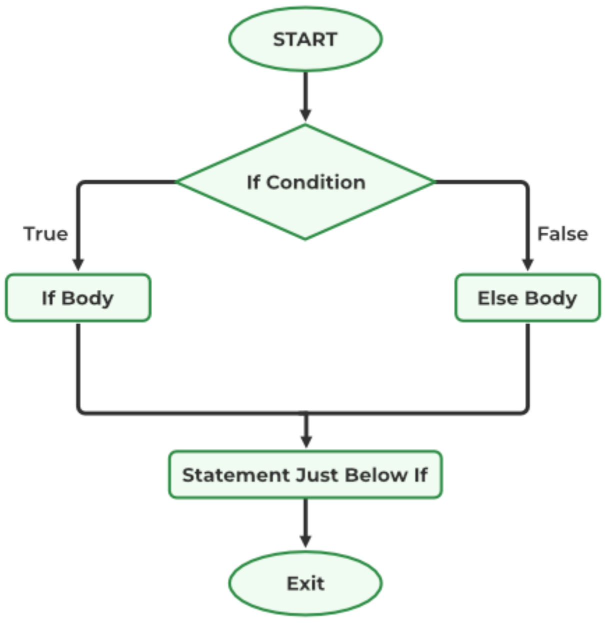


Fig. 2.2: Flow Diagram of if else

Example of if-else

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
    int i = 20;
    if (i < 15) {
        printf("i is smaller than 15");
    }
    else {
        printf("i is greater than 15");
    }
    return 0;
}
```

Output

```
i is greater than 15
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

3. Nested if-else statement

A nested if in C/C++ is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, C/C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax of Nested if-else

```
if(condition1)
{
  //Executes when condition1 is true
  if(condition_2)
  {
    //statement1
  }
  else
  {
      //Statement2
   }
}
else
{
    if(condition_3)
  {
    //statement3
  }
  else
  {
      //Statement4
   }
}
```

Example of Nested if-else

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");
        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }
    else {
        if (i == 20) {

            // Nested - if statement
            // Will only be executed if statement above
            // is true
            if (i < 22)
            printf("i is smaller than 22 too\n");
         else
            printf("i is greater than 25");
      }
   }

   return 0;
}
```

Output

```
i is smaller than 15

i is smaller than 12 too
```

4. if-else-if Ladder in C/C++
The if-else-if statements are used when the user has to decide among multiple options. The C/C++ if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C/C++ else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Syntax of if-else-if Ladder

```
if(condition)
   statement;
else if (condition)
   statement;
.
.
else
   statement;
```

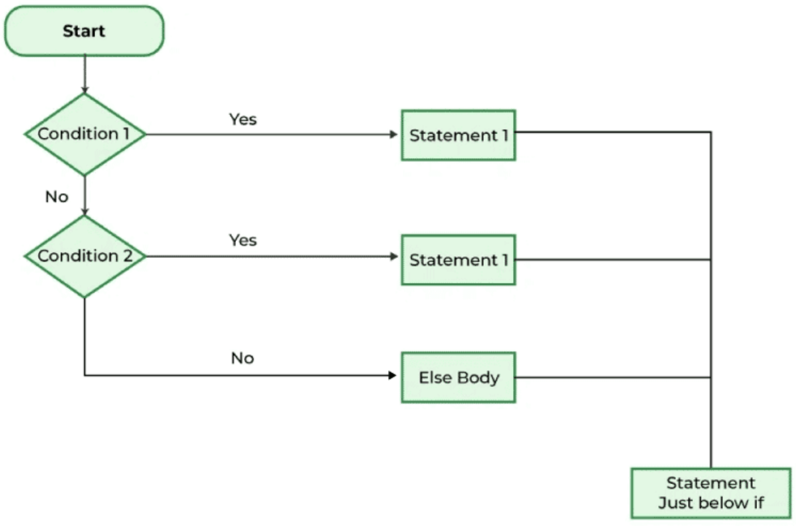


Fig 2.3: Flow Diagram of if-else-if

Example of if-else-if Ladder

```
// C program to illustrate nested-if statement
#include <stdio.h>
```

```
int main()
{
    int i = 20;
    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

Output

```
i is 20
```

5. switch Statement in C/C++

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Syntax of switch

```
switch(expression)
{
    case value1:
        statements;
    case value2:
        statements;
    ....
    ....
    ....
    default:
        statements;
}
```

Note: *The switch expression should evaluate to either integer or character. It cannot evaluate any other data type.*

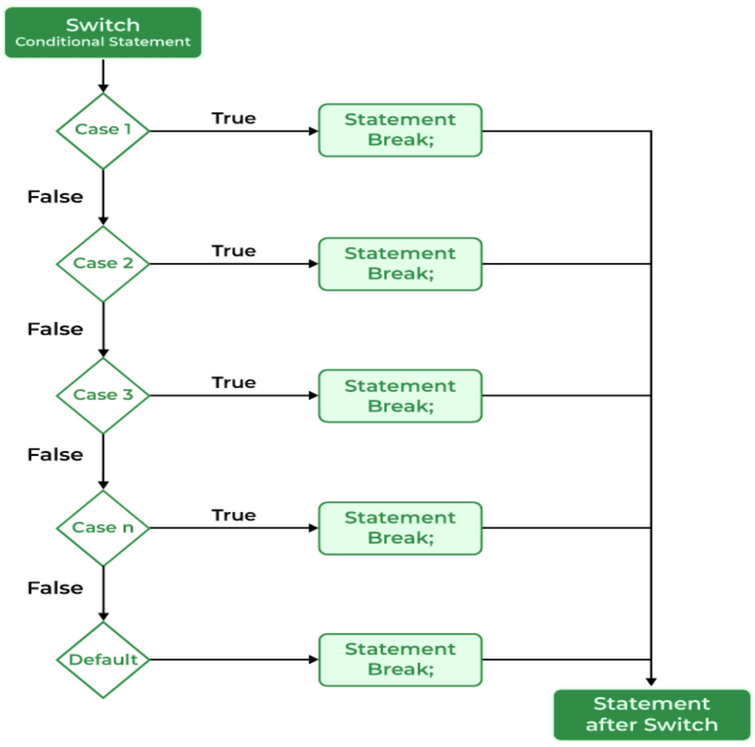


Fig 2.4: Flowchart of switch in C

Example of switch Statement

```
// C Program to illustrate the use of switch statement
#include <stdio.h>

int main()
{
   // variable to be used in switch statement
   int var = 2;
   // declaring switch cases
   switch (var) {
   case 1:
       printf("Case 1 is executed");
       break;
   case 2:
       printf("Case 2 is executed");
       break;
   default:
       printf("Default Case is executed");
       break;
   }

   return 0;
}
```

Output

```
Case 2 is executed
```

```
0: -25
```

6. Jump Statements in C/C++

These statements are used in C/C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

A) break

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Syntax of break

```
break;
```

Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.

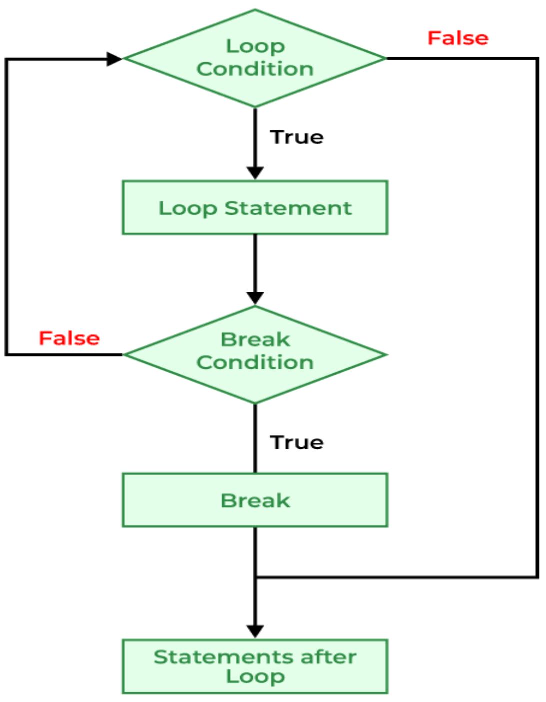


Fig 2.5 flowchart for break statement

Example of break

```
// C program to illustrate
// to show usage of break
// statement
#include <stdio.h>

void findElement(int arr[], int size, int key)
{
   // loop to traverse array and search for key
   for (int i = 0; i < size; i++) {
      if (arr[i] == key) {
         printf("Element found at position: %d", (i + 1));
         break;
      }
```

```
    }
}

int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    // no of elements
    int n = 6;
    // key to be searched
    int key = 3;
    // Calling function to find the key
    findElement(arr, n, key);
    return 0;
}
```

Output

```
Element found at position: 3
```

B) continue

This loop control statement is just like the break statement. The continue statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

Syntax of continue

```
continue;
```

Example of continue

```
// C program to explain the use
// of continue statement
#include <stdio.h>
int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;
        else
            // otherwise print the value of i
            printf("%d ", i);
    }

    return 0;
}
```

Output

```
1 2 3 4 5 7 8 9 10
```

If you create a variable in if-else in C/C++, it will be local to that if/else block only. You can use global variables inside the if/else block. If the name of the variable you created in if/else is as same as any global variable then priority will be given to the `local variable`.

```
#include <stdio.h>
int main()
{
   int gfg = 0; // local variable for main
   printf("Before if-else block %d\n", gfg);
   if (1) {
      int gfg = 100; // new local variable of if block
      printf("if block %d\n", gfg);
   }
   printf("After if block %d", gfg);
   return 0;
}
```

Output

```
Before if-else block 0

if block 100

After if block 0
```

C) goto

The goto statement in C/C++ also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Examples of goto

```
// C program to print numbers
// from 1 to 10 using goto
// statement
#include <stdio.h>
// function to print numbers from 1 to 10
void printNumbers()
{
   int n = 1;
label:
   printf("%d ", n);
   n++;
   if (n <= 10)
      goto label;
}

// Driver program to test above function
int main()
{
   printNumbers();
   return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

C++ Loops:

In C++ programming, sometimes there is a need to perform some operation **more than once** or (say) **n number** of times. **For example,** suppose we want to print "Hello World" 5 times. Manually, we have to write **cout** for the C++ statement 5 times as shown.

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

int main() {
     // Print hello world 5 times
   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
```

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. In this case, loops come into use allowing users to repeatedly execute a block of statements.

```
#include <iostream>
using namespace std;
int main() {
     for (int i = 0; i < 5; i++) {
       cout << "Hello World\n";
   }
   return 0;
}
```

Output

```
Hello World

Hello World
Hello World

Hello World

Hello World
```

What are Loops?

In C++ programming, a loop is an instruction that is used to repeatedly execute a code until a certain condition is reached. They are useful for performing repetitive tasks without having to write the same code multiple times.

Loops can be classified on the basis of the sequency in which they check the condition relative to the execution of the associated block of code.

1. **Entry Controlled loops**: In this type of loop, the test condition is tested before entering the loop body.
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.

C++ provides three loops:

- for Loop
- while Loop
- do-while Loop

Let's understand each of them in detail.

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. It is an entry-controlled loop that enables us to perform n number of steps together in one line.

Syntax

for (initialization; condition; updation) *{*
// body of for loop
}

The various **parts of the for loop** are:

- **Initialization**: Initialize the loop variable to some initial value.
- **Test Condition**: This specifies the test condition. If the condition evaluates to true, then body of the loop is executed, and loop variable is updated according to update expression. If evaluated false, loop is terminated.
- **Update Expression**: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example

```
#include <iostream>
using namespace std;
int main() {
   // For loop that starts with i = 1 and ends
   // when i is greater than 5.
  for (int i = 1; i <= 5; i++) {
    cout << i << " ";
  }

  return 0;
}
```

Output

```
1 2 3 4 5
```

Important Properties of for Loop

- The initialization and updation statements can perform operations unrelated to the condition statement, or nothing at all – if you wish to do. But the good practice is to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the for loop and will be released out of the loop.
- Don't forget that the variable which was declared in the initialization statement can be modified during the loop, as well as the variable checked in the condition.

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 Loop is used in situations where we do not know the exact number of iterations of the loop beforehand. It is an entry-controlled loop whose execution is terminated on the basis of the test conditions.

Syntax

while (condition) *{*
// Body of the loop
update expression
}

Example

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

int main() {
     // Initialization
     int i = 1;
     // while loop that starts with i = 1 and ends
     // when i is greater than 5.
   while (i <= 5) {
       cout << i << " ";
         // Updation
         i++;
   }

   return 0;
}
```

Output

```
1 2 3 4 5
```

do-while Loop:

The do-while Loop is also a loop whose 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. So, in a do-while loop, the loop body will ***execute at least once*** irrespective of the test condition.

Syntax

***do** {*
// Body of the loop
// Update expression
} ***while (condition);***

Example

```
#include <iostream>
using namespace std;
int main() {
     // Initialization
     int i = 1;
     // while loop that starts with i = 1 and ends
     // when i is greater than 5.
    do {
       cout << i << " ";
         // Updation
         i++;
   }while (i <= 5);

   return 0;
}
```

Output

```
1 2 3 4 5
```

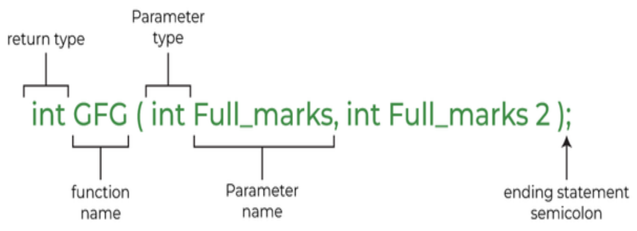
Remember the last semicolon at the end of the loop.

3. STRUCTURES AND FUNCTIONS

Functions in C++:

A function is a set of statements that takes input, does some specific computation, and produces output. The idea is to put some commonly or repeatedly done tasks together to make a **function** so that instead of writing the same code again and again for different inputs, we can call this function.
In simple terms, a function is a block of code that runs only when it is called.

Syntax:


return type
Parameter type
int GFG (int Full_marks, int Full_marks 2);
function name
Parameter name
ending statement semicolon

Example:

```
// C++ Program to demonstrate working of a function
#include <iostream>
using namespace std;

// Following function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}
// main function that doesn't receive any parameter and
// returns integer
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
```

Why Do We Need Functions?

- Functions help us in ***reducing code redundancy***. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to make changes in only one place if we make changes to the functionality in future.

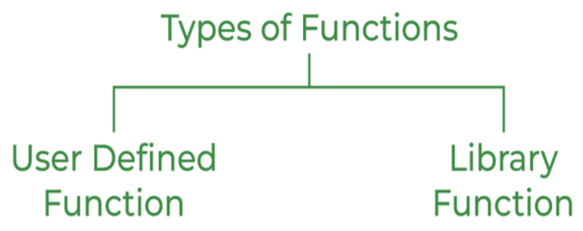
- Functions make code ***modular***. Consider a big file having many lines of code. It becomes really simple to read and use the code, if the code is divided into functions.
- Functions provide ***abstraction***. For example, we can use library functions without worrying about their internal work.

Function Declaration:
A function declaration tells the compiler about the number of parameters, data types of parameters, and returns type of function. Writing parameter names in the function declaration is optional but it is necessary to put them in the definition. Below is an example of function declarations. (parameter names are not present in the below declarations).
Example:

```
// C++ Program to show function that takes
// two integers as parameters and returns
// an integer
int max(int, int);

// A function that takes an int
// pointer and an int variable
// as parameters and returns
// a pointer of type int
int* swap(int*, int);
```

Types of Functions:

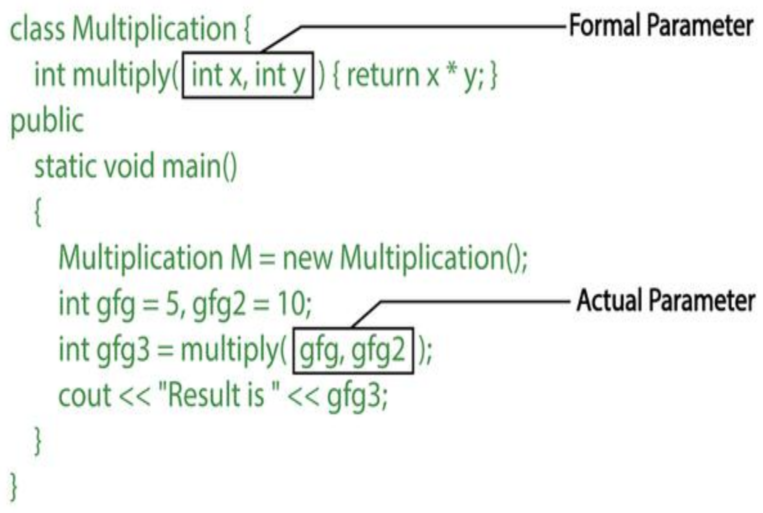


User Defined Function
User-defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as "***tailor-made functions***" which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

Library Function
Library functions are also called "***built-in Functions***". These functions are part of a compiler package that is already defined and consists of a special function with special and different meanings. Built-in Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.
For Example: sqrt(), setw(), strcat(), etc.

Parameter Passing to Functions:
The parameters passed to the function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.

The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.



```
class Multiplication {
  int multiply( int x, int y ) { return x * y; }
public
  static void main()
  {
    Multiplication M = new Multiplication();
    int gfg = 5, gfg2 = 10;
    int gfg3 = multiply( gfg, gfg2 );
    cout << "Result is " << gfg3;
  }
}
```

There are two most popular ways to pass parameters:

1. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function's formal parameters. The actual and formal parameters are stored in different memory locations, so any changes made in the functions are not reflected in the actual parameters of the caller.

2. ***Pass by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are reflected in the actual parameters of the caller.

Function Definition:

Pass by value is used where the value of x is not modified using the function fun().

```
// C++ Program to demonstrate function definition
#include <iostream>
using namespace std;
void fun(int x)
{
   // definition of
   // function
   x = 30;
}
int main()
{
   int x = 20;
   fun(x);
   cout << "x = " << x;
   return 0;
}
```

Output

```
x = 20
```

Functions Using Pointers:
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', the 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.

```
// C++ Program to demonstrate working of
// function using pointers
#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
```

Table 3.1 Difference between call by value and call by reference in C++

Call by value	**Call by reference**
A copy of the value is passed to the function	An address of value is passed to the function
Changes made inside the function are not reflected on other functions	Changes made inside the function are reflected outside the function as well
Actual and formal arguments will be created at different memory location	Actual and formal arguments will be created at same memory location.

Points to Remember About Functions in C++
1. Most C++ program has a function called main() that is called by the 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 the return statement in the body of the 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. 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.

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.
Recursion:
When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

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

void directRecursion(int n) {
   if (n > 0) {
      cout << n << " ";
      directRecursion(n - 1); // Function calls itself
   }
}

int main() {
   directRecursion(10);
   return 0;
}
```

Output

```
5 4 3 2 1
```

Types of overloading in C++ are:
- *Function overloading*
- *Operator overloading*

Function Overloading
Function Overloading is defined as the process of having two or more functions with the same name, but different parameters. In function overloading, the function is redefined by using either different types or number of arguments. It is only through these differences a compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don’t need to use different names for the same action.

Example: changing number of arguments of add() method

```
// program of function overloading when number of arguments
// vary
#include <iostream>
using namespace std;
class Cal {
public:
    static int add(int a, int b) { return a + b; }
    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void)
{
    Cal C; // class object declaration.
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}
```

Output

```
30

55
```

Example: when the type of the arguments varies.

```
// Program of function overloading with different types of
// arguments.
#include <iostream>
using namespace std;
int mul(int, int);
float mul(float, int);

int mul(int a, int b) { return a * b; }
float mul(double x, int y) { return x * y; }
int main()
{
    int r1 = mul(6, 7);
    float r2 = mul(0.2, 3);
    cout << "r1 is : " << r1 << endl;
    cout << "r2 is : " << r2 << endl;
    return 0;
}
```

Output

```
r1 is : 42
```

```
r2 is : 0.6
```

Structures, Unions and Enumerations in C++:

Structures, unions and enumerations (enums) are 3 user defined data types in C++. User defined data types allow us to create a data type specifically tailored for a particular purpose. It is generally created from the built-in or derived data types. Let’s take a look at each of them one by one.

Structure:
In C++, strcuture is a user-defined data type that is used to combine data of different types. It is similar to an array but unlike an array, which stores elements of the same type, a structure can store elements of different data types. C++ structures can also have member functions to manipulate its data.

Create Structure

A structure has to be defined before being usable in the program. It is defined using **struct** keyword.

```
struct structure_name{
   type1 member1;
   type2 member2;
   .
   .
   typeN memberN;
};
```

This definition does not allocate any memory to the structure. We have to crate structure variables separately to use it.

```
structure_name var_name;
```

We can also assign some values to the members:

```
struct structure_name = {val1, val2, ..., valN};
```

Access and Update

Structure members can be accessed using the dot operator(.)

```
struct structure_name;
// Accessing first member
structure_name.member1;
// Accessing second member
structure_name.member2;
// Accessing third member
structure_name.member3;
```

Example

```
 {...}
// Define structure
struct GFG {
   int G1;
   char G2;
   float G3;
};
int main() {
   // Create object of structure
   GFG Geek = {85, 'G', 989.45};
   // Accessing structure members values
   cout << Geek.G1 << endl;
   cout << Geek.G2 << endl;
   cout << Geek.G3;

 {...}
```

Output

```
85

G

989.45
```

Explanation: In the above code, values: **(85, ‘G’, 989.45)** are assigned to the G1, G2, and G3 member variables of the structure GFG, and these values are printed at the end using dot (.) operator.

Union:

In C++, union is a user-defined datatype in which we can define members of different types of data types just like structures but unlike a structure, where each member has its own memory, a union member shares the same memory location.

Create Union

Union is first defined using **union** keyword:

```
union union_name{
   type1 member1;
   type2 member2;
   .
   .
   typeN memberN;
};
```

Then we can create union variables:

```
union_name var_name;
```

Access and Update

Only one member of a union stores memory at one time.

```
var_name.member1 = val
```

Example

```
 {...}
// Defining a Union
union GFG {
   int G1;
   char G2;
   float G3;
};
int main() {

   // Create an object of GFG union
   GFG Geek;
   // Assign union's member variables
   Geek.G1 = 85;
   // Accessing union members values
   cout << Geek.G1 << endl;
   Geek.G2 = 'G';
   cout << Geek.G2 << endl;
   Geek.G3 = 989.45;
   cout <<  Geek.G3;

 {...}
```

Output

```
85

G

989.45
```

Enumeration:
In C++, enumeration (enum) is a user-defined type that consists of a set of named integral constants. Enumerations help make the code more readable and easier to maintain by assigning meaningful names to constants.

Create Enums

Just like all other user defined data types, **enums** also needs to be defined before we can use it.

```
enum enum_name {
   value1, value2, value3…..valueN
};
```

Once defined, it can be used in the C++ program.

```
enum_name var_name = value
```

This value should be taken from the defined value.

Example

```
 {...}
   // Defining enum Gender
   enum GFG { Male, Female, };

   // Creating GFG type variable and assigning
   // value
   GFG Geek = Male;
   switch (Geek) {
   case Male:
      cout << "Who is he?";
      break;
   case Geek2:
      Female << "Who is she?";
      break;
   default:
      cout << "Who is they?";
   }

 {...}
```

Output

```
Belongs to GFG
```

4. CLASSES AND OBJECTS

C++ Classes and Objects:

In C++, classes and objects are the basic building **block that leads to Object-Oriented programming in C++.**
A class 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 brands but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range,* etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables 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 be *applying brakes*, *increasing speed*, etc.

But we cannot use the class as it is. We first have to create an object of the class to use its features. An **Object** is an instance of a Class.
Note: *When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.*

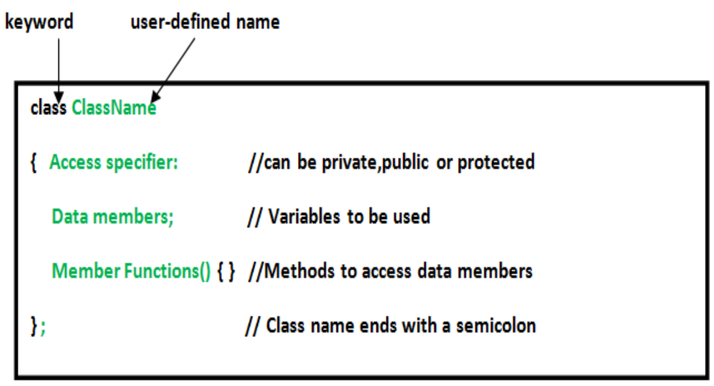
Defining Class in C++

A class is defined in C++ using the keyword c**lass** followed by the name of the class. The following is the syntax:

```
class ClassName {
    access_specifier:
    // Body of the class
};
```

Here, the access specifier defines the level of access to the class's data members.
Example

```
class ThisClass {
    public:
    int var;     // data member
    void print() {          // member method
        cout << "Hello";
     }
};
```


keyword
user-defined name
class ClassName
{ Access specifier: //can be private,public or protected
Data members; // Variables to be used
Member Functions() { } //Methods to access data members
}; // Class name ends with a semicolon

What is an Object in C++?

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 to Create an Object

We can create an object of the given class in the same way we declare the variables of any other inbuilt data type.

```
ClassName ObjectName;
```

Example

```
MyClass obj;
```

In the above statement, the object of MyClass with name obj is created.

Accessing Data Members and Member Functions

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

```
obj.printName()
```

Example of Class and Object in C++

The below program shows how to define a simple class and how to create an object of it.

```
// C++ program to illustrate how create a simple class and
// object
#include <iostream>
#include <string>
using namespace std;
// Define a class named 'Person'
class Person {
public:
    // Data members
    string name;
    int age;
    // Member function to introduce the person
    void introduce()
    {
        cout << "Hi, my name is " << name << " and I am "
             << age << " years old." << endl;
    }
};
```

```
int main()
{
   // Create an object of the Person class
   Person person1;
   // accessing data members
   person1.name = "Alice";
   person1.age = 30;
   // Call the introduce member method
   person1.introduce();
   return 0;
}
```

Output

```
Hi, my name is Alice and I am 30 years old.
```

Access Modifiers:
In C++ classes, we can control the access to the members of the class using Access Specifiers. Also known as access modifier, they are the keywords that are specified in the class and all the members of the class under that access specifier will have particular access level.
In C++, there are 3 access specifiers that are as follows:

1. **Public:** Members declared as public can be accessed from outside the class.
2. **Private:** Members declared as private can only be accessed within the class itself.
3. **Protected:** Members declared as protected can be accessed within the class and by derived classes.

If we do not specify the access specifier, the private specifier is applied to every member by default.

Example of Access Specifiers

```
// C++ program to demonstrate accessing of data members
#include <bits/stdc++.h>
using namespace std;
class Geeks {
private:
   string geekname;
   // Access specifier
public:
   // Member Functions()
   void setName(string name) { geekname = name; }

   void printname() { cout << "Geekname is:" << geekname; }
};
int main()
{
   // Declare an object of class geeks
   Geeks obj1;
   // accessing data member
   // cannot do it like: obj1.geekname = "Abhi";
   obj1.setName("Abhi");
   // accessing member function
   obj1.printname();
   return 0;
}
```

Output

```
Geekname is:Abhi
```

In the above example, we cannot access the data member geekname outside the class. If we try to access it in the main function using dot operator, obj1.geekname, then program will throw an error.

Member Function in C++ Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Till now, we have defined the member function inside the class, but we can also define the member function outside the class. To define a member function outside the class definition,

- We have to first declare the function prototype in the class definition.
- Then we have to use the **scope resolution operator (::)** along with the class name and function name.

Example

```
// C++ program to demonstrate member function
// definition 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 the keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calls 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 **4 types of constructors in C++ classes:**

- Default Constructors: The constructor that takes no argument is called default constructor.
- Parameterized Constructors: This type of constructor takes the arguments to initialize the data members.
- Copy Constructors: Copy constructor creates the object from an already existing object by copying it.

Example of Constructor

```
// 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;

   // obj2 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
```

Note: *If the programmer does not define the constructor, the compiler automatically creates the default, copy constructor.*

Destructors:

Destructor is another special member function that is called by the compiler when the scope of the object ends. It deallocates all the memory previously used by the object of the class so that there will be no memory leaks. The destructor also have the same name as the class but with tilde(~) as prefix.

Example of Destructor

```
// 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
```

UNIT-2

5. MEMBER FUNCTIONS
6.

Access Modifiers in C++:

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as **Data Hiding**. Consider a real-life example: The Research and Analysis Wing (R&AW), having 10 core members, has come into possession of sensitive confidential information regarding national security. Now we can correlate these core members to data members or member functions of a class, which in turn can be correlated to the R&A Wing. These 10 members can directly access the confidential information from their wing (the class), but anyone apart from these 10 members can't access this information directly, i.e., outside functions other than those prevalent in the class itself can't access the information (that is not entitled to them) without having either assigned privileges (such as those possessed by a friend class or an inherited class) or access to one of these 10 members who is allowed direct access to the confidential information (similar to how private members of a class can be accessed in the outside world through public member functions of the class that have direct access to private members). This is what data hiding is in practice.
Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be **Private**. Understanding how to use public, private, and protected access modifiers is essential.
Let us now look at each one of these access modifiers in detail:
1. Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
Example:

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
using namespace std;
// class definition
class Circle
{
   public:
      double radius;

      double  compute_area()
      {
         return 3.14*radius*radius;
```

```
        }
};

// main function
int main()
{
    Circle obj;
    // accessing public datamember outside class
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5
Area is: 94.985
```

In the above program, the data member *radius* is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

2. Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend function/friend class are allowed to access the private data members of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
        double  compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
        }
};

// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;
    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Output:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
         double radius;
                ^
31:9: error: within this context
     obj.radius = 1.5;
         ^
```

The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error. However, we can access the private data members of a class indirectly using the public member functions of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
        void compute_area(double r)
        {   // member function can access private
            // data member radius
            radius = r;
            double area = 3.14*radius*radius;
            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);
    return 0;
}
```

Output:

```
Radius                                          is:                                        1.5
Area is: 7.065
```

3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of inheritance.

Example:

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;
// base class
class Parent
{
   // protected data members
   protected:
   int id_protected;
};

// sub class or derived class from public base class
class Child : public Parent
{
   public:
   void setId(int id)
   {
      // Child class is able to access the inherited
      // protected data members of base class
      id_protected = id;
   }

   void displayId()
   {
      cout << "id_protected is: " << id_protected << endl;
   }
};

// main function
int main() {
   Child obj1;
   // member function of the derived class can
   // access the protected data members of the base class
   obj1.setId(81);
   obj1.displayId();
   return 0;
}
```
Output:
```
id_protected is: 81
```

Friend Class and Function in C++:

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.
We can declare a friend class in C++ by using the **friend** keyword.
Syntax:
```
friend class class_name;    // declared in the base class
```

Example:
```
// C++ Program to demonstrate the
// functioning of a friend class
#include <iostream>
using namespace std;

class GFG {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    GFG()
    {
        private_variable = 10;
        protected_variable = 99;
    }
    // friend class declaration
    friend class F;
};

// Here, class F is declared as a
// friend inside class GFG. Therefore,
// F is a friend of class GFG. Class F
// can access the private members of
// class GFG.
class F {
public:
    void display(GFG& t)
    {
        cout << "The value of Private Variable = "
             << t.private_variable << endl;
        cout << "The value of Protected Variable = "
             << t.protected_variable;
    }
};

// Driver code
int main()
{
    GFG g;
    F fri;
    fri.display(g);
    return 0;
}
```

Output

```
The value of Private Variable = 10

The value of Protected Variable = 99
```

Note: *We can declare friend class or function anywhere in the base class body whether its private, protected or public block. It works all the same.*

Friend Function:

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are not the member functions of the class but can access and manipulate the private and protected members of that class for they are declared as friends.

A friend function can be:

1. **A global function**
2. **A member function of another class**

Syntax:

```
friend return_type function_name (arguments);    // for a global function
            or
friend return_type class_name::function_name (arguments);    // for a member function of
another class
```

1. Global Function as Friend Function

We can declare any global function as a friend function. The following example demonstrates how to declare a global function as a friend function in C++:

Example:

```
// C++ program to create a global function as a friend
// function of some class
#include <iostream>
using namespace std;
class base {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }

      // friend function declaration
    friend void friendFunction(base& obj);
};

// friend function definition
void friendFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}
// driver code
int main()
{
    base object1;
    friendFunction(object1);

    return 0;
}
```

Output

```
Private Variable: 10

Protected Variable: 99
```

In the above example, we have used a global function as a friend function. In the next example, we will use a member function of another class as a friend function.

2. Member Function of Another Class as Friend Function

We can also declare a member function of another class as a friend function in C++. The following example demonstrates how to use a member function of another class as a friend function in C++:

Example:

```
// C++ program to create a member function of another class
// as a friend function
#include <iostream>
using namespace std;
class base; // forward definition needed
// another class in which function is declared
class anotherClass {
public:
    void memberFunction(base& obj);
};
// base class for which friend is declared
class base {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }
    // friend function declaration
    friend void anotherClass::memberFunction(base&);
};

// friend function definition
void anotherClass :: memberFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}
// driver code
int main()
{
    base object1;
    anotherClass object2;
    object2.memberFunction(object1);
    return 0;
}
```

Output

```
Private Variable: 10

Protected Variable: 99
```

***Note**: The order in which we define the friend function of another class is important and should be taken care of. We always have to define both the classes before the function definition. Thats why we have used out of class member function definition.*

Features of Friend Functions

- A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to **access** the **private and protected data** of a class.
- A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword "**friend**" inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword "friend" is placed only in the function declaration of the friend function and **not** in the **function definition or call.**
- A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be declared in any section of the class i.e. public or private or protected.

Advantages of Friend Functions:

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions:

- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

C++ Static Data Members:

Static data members are class members that are declared using **static** keywords. A static member has certain special characteristics which are as follows:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts outside the class itself.
- It is visible can be controlled with the class access specifiers.
- Its lifetime is the entire program.

Syntax

```
className {
    static data_type data_member_name;
    .....
}
```

Static data members are useful for maintaining data shared among all instances of a class.

Example

Below is the C++ program to demonstrate the working of static data members:

```
// C++ Program to demonstrate the use of
// static data members
#include <iostream>
using namespace std;
// class definition
class A {
public:
   // static data member here
   static int x;
   A() { cout << "A's constructor called " << endl; }
};
// we cannot initialize the static data member inside the
// class due to class rules and the fact that we cannot
// assign it a value using constructor
int A :: x = 2;
// Driver code
int main()
{
   // accessing the static data member using scope
   // resultion operator
```

```
    cout << "Accessing static data member: " << A::x
         << endl;
    return 0;
}
```

Output

```
Accessing static data member: 2
```

Defining Static Data Member

As told earlier, the static members are only declared in the class declaration. If we try to access the static data member without an explicit definition, the compiler will give an error.

To access the static data member of any class we have to define it first and static data members are defined outside the class definition. The only exception to this are static const data members of integral type which can be initialized in the class declaration.

Syntax

```
datatype class_name::var_name = value...;
```

For example, in the above program, we have initialized the static data member using the following statement:

```
int A::x = 10
```

Note: *The static data members are initialized at compile time so the definition of static members should be present before the compilation of the program*

Accessing a Static Member:

We can access the static data member without creating the instance of the class. Just remember that we need to initialize it beforehand. There are 2 ways of accessing static data members:

1. **Accessing static data member using Class Name and Scope Resolution Operator**

The class name and the scope resolution operator can be used to access the static data member even when there are no instances/objects of the class present in the scope.

Syntax

```
Class_Name :: var_name
```

Example

```
A::x
```

2. **Accessing static data member through Objects**

We can also access the static data member using the objects of the class using dot operator.

Syntax

```
object_name . var_name
```

Example

```
obj.x
```

Note: *The access to the static data member can be controlled by the class access modifiers.*

Example to Verify the Properties of the Static Data Members

The below example verifies the properties of the static data member that are told above:

```
// C++ Program to demonstrate
// the working of static data member
#include <iostream>
using namespace std;
// creating a dummy class to define the static data member
// it will inform when its type of the object will be
// created
class stMember {
```

```
public:
    int val;
    // constructor to inform when the instance is created
    stMember(int v = 10): val(v) {
        cout << "Static Object Created" << endl;
    }
};
// creating a demo class with static data member of type
// stMember
class A {
public:
    // static data member
    static stMember s;
    A() { cout << "A's Constructor Called " << endl; }
};

stMember A::s = stMember(11);
// Driver code
int main()
{

    // Statement 1: accessing static member without creating
    // the object
    cout << "accessing static member without creating the "
            "object: ";
    // this verifies the independency of the static data
    // member from the instances
    cout << A::s.val << endl;
    cout << endl;

    // Statement 2: Creating a single object to verify if
    // the seperate instance will be created for each object
    cout << "Creating object now: ";
    A obj1;
    cout << endl;

    // Statement 3: Creating multiple objects to verify that
    // each object will refer the same static member
    cout << "Creating object now: ";
    A obj2;
    cout << "Printing values from each object and classname"
         << endl;

    cout << "obj1.s.val: " << obj1.s.val << endl;
    cout << "obj2.s.val: " << obj2.s.val << endl;
    cout << "A::s.val: " << A::s.val << endl;

    return 0;
}
```

Output

```
Static Object Created
accessing static member without creating the object: 11

Creating object now: A's Constructor Called

Creating object now: A's Constructor Called
Printing values from each object and classname
obj1.s.val: 11
obj2.s.val: 11
A::s.val: 11
```

Static Member Function in C++:

Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.
- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- Static member functions have a scope inside the class and cannot access the current object pointer.
- You can also use a static member function to determine how many objects of the class have been created.

The reason we need Static member function:

- Static members are frequently used to store information that is shared by all objects in a class.
- For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

Example:

```
// C++ Program to show the working of
// static member functions
#include <iostream>
using namespace std;
class Box
{
   private:
    static int length;
    static int breadth;
    static int height;
    public: static void print()
    {
        cout << "The value of the length is: " << length << endl;
        cout << "The value of the breadth is: " << breadth << endl;
        cout << "The value of the height is: " << height << endl;
    }
};
// initialize the static data members
int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;
// Driver Code
int main()
{
    Box b;
    cout << "Static member function is called through Object name: \n" << endl;
    b.print();
    cout << "\nStatic member function is called through Class name: \n" << endl;
    Box::print();
    return 0;
}
```

Output

```
Static member function is called through Object name:
The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30
Static member function is called through Class name:
The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30
```

Operator Overloading in C++:

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Example:

```
int a;
float b,sum;
sum = a + b;
```

Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

Implementation:

```
// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading
class A {
    statements;
};

int main()
{
    A a1, a2, a3;
    a3 = a1 + a2;
    return 0;
}
```

In this example, we have 3 variables "a1", "a2" and "a3" of type "class A". Here we are trying to add two objects "a1" and "a2", which are of user-defined type i.e. of type "class A" using the "+" operator. This is not allowed, because the addition operator "+" is predefined to operate only on built-in data types. But here, "class A" is a user-defined type, so the compiler generates an error. This is where the concept of "Operator overloading" comes in.

Now, if the user wants to make the operator "+" add two class objects, the user has to redefine the meaning of the "+" operator such that it adds two class objects. This is done by using the concept of "Operator overloading". So the main idea behind "Operator overloading" is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

Example of Operator Overloading in C++

```
// C++ Program to Demonstrate
// Operator Overloading
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

Difference between Operator Functions and Normal Functions
Operator functions are the same as normal functions. The only differences are that the name of an operator function is always the **operator keyword** followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.

Example

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    void print() { cout << real << " + i" << imag << endl; }
    // The global operator function is made friend of this
    // class so that it can access private members
    friend Complex operator+(Complex const& c1,
                    Complex const& c2);
};
Complex operator+(Complex const& c1, Complex const& c2)
{
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3
      = c1
        + c2; // An example call to &quot;operator+&quot;
   c3.print();
   return 0;
}
```

Output

```
12 + i9
```

Can We Overload All Operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

```
sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member operator))
Ternary or conditional (?:)
```

Operators that can be Overloaded in C++

We can overload

- ***Unary operators***
- ***Binary operators***
- ***Special operators*** *([ ], (), etc)*

Table 5.1 Operators that can be overloaded

Operators that can be overloaded	**Examples**
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^

Operators that can be overloaded	**Examples**
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	( )
Logical	&, | |, !
Relational	>, < , = =, <=, >=

Why can't the above-stated operators be overloaded?

1. sizeof Operator

This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

2. typeid Operator

This provides a CPP program with the ability to recover the actually derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

3. Scope resolution (::) Operator

This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

4. Class member access operators (.(dot), .* (pointer to member operator))

The importance and implicit use of class member access operators can be understood through the following example:

Example:

```
// C++ program to demonstrate operator overloading
// using dot operator
#include <iostream>
using namespace std;
class ComplexNumber {
private:
   int real;
   int imaginary;
public:
   ComplexNumber(int real, int imaginary)
   {
      this->real = real;
      this->imaginary = imaginary;
   }
   void print() { cout << real << " + i" << imaginary; }
   ComplexNumber operator+(ComplexNumber c2)
   {
      ComplexNumber c3(0, 0);
      c3.real = this->real + c2.real;
      c3.imaginary = this->imaginary + c2.imaginary;
      return c3;
   }
};
int main()
{
   ComplexNumber c1(3, 5);
   ComplexNumber c2(2, 4);
   ComplexNumber c3 = c1 + c2;
   c3.print();
   return 0;
}
```

Output

```
5 + i9
```

Explanation:
The statement ComplexNumber c3 = c1 + c2; is internally translated as ComplexNumber c3 = c1.operator+ (c2); in order to invoke the operator function. The argument c1 is implicitly passed using the **'.'** operator. The next statement also makes use of the dot operator to access the member function print and pass c3 as an argument.
Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

5. Ternary or conditional (?:) Operator

The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

```
conditional statement ? expression1 (if statement is TRUE) : expression2 (else)
```

A function overloading the ternary operator for a class say ABC using the definition

```
ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);
```

would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.
Important Points about Operator Overloading:
1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor).
3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.
Example:

```
// C++ Program to Demonstrate the working
// of conversion operator
#include <iostream>
using namespace std;
class Fraction {
private:
    int num, den;

public:
    Fraction(int n, int d)
    {
        num = n;
        den = d;
    }

    // Conversion operator: return float value of fraction
    operator float() const
    {
        return float(num) / float(den);
    }
};

int main()
{
    Fraction f(2, 5);
    float val = f;
    cout << val << '\n';
    return 0;
}
```

Output

```
0.4
```

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.
4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.
Example:

```
// C++ program to demonstrate can also be used for implicit
// conversion to the class being constructed
#include <iostream>
using namespace std;
class Point {
private:
    int x, y;

public:
    Point(int i = 0, int j = 0)
    {
        x = i;
        y = j;
    }
    void print()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main()
{
    Point t(20, 20);
    t.print();
    t = 30; // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output

```
x = 20, y = 20

x = 30, y = 0
```

6. INHERITANCE

Inheritance in C++:

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 in C++.

Syntax of Inheritance in C++

```
class  derived_class_name : access-specifier  base_class_name
{
      //    body ....
};
```

where,

- **class:** keyword to create a new class
- **derived_class_name**: name of the new class, which will inherit the base class
- **access-specifier**: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.
- **base-class-name**: name of the base class.

Note*: A derived class doesn't inherit* ***access*** *to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.*

Example:

```
class ABC : private XYZ {...}          //  private derivation
class ABC : public XYZ {...}           //  public derivation
class ABC : protected XYZ {...}      //  protected derivation
class ABC: XYZ {...}                   //  private derivation by default
```

The following programs demonstrate how to implement inheritance in our C++ programs.

Example 1: Program to Demonstrate the Simple Inheritance of a Class

```
// C++ program to demonstrate how to inherit a class
#include <iostream>
using namespace std;
// Base class that is to be inherited
class Parent {
public:
   // base class members
   int id_p;
   void printID_p()
   {
      cout << "Base ID: " << id_p << endl;
   }
};
// Sub class or derived publicly inheriting from Base
// Class(Parent)
class Child : public Parent {
public:
   // derived class members
   int id_c;
   void printID_c()
```

```
    {
        cout << "Child ID: " << id_c << endl;
    }
};
// main function
int main()
{
    // creating a child class object
    Child obj1;
    // An object of class child has all data members
    // and member functions of class parent
    // so we try accessing the parents method and data from
    // the child class object.
    obj1.id_p = 7;
    obj1.printID_p();
    // finally accessing the child class methods and data
    // too
    obj1.id_c = 91;
    obj1.printID_c();
    return 0;
}
```

Output

```
Base ID: 7

Child ID: 91
```

In the above program, the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

Example 2: Access the Inherited Members of the Base Class in Derived Class

```
// C++ program to illustrate how to access the inherited
// members of the base class in derived class
#include <iostream>
using namespace std;
// Base class
class Base {
public:
    // data member
    int publicVar;
    // member method
    void display()
    {
        cout << "Value of publicVar: " << publicVar;
    }
};

// Derived class
class Derived : public Base {
public:
    // Function to display inherited member
    void displayMember()
    {
        // accessing public base class member method
        display();
    }
    // Function to modify inherited member
    void modifyMember(int pub)
    {
        // Directly modifying public member
        publicVar = pub;
    }
};

int main()
{
    // Create an object of Derived class
    Derived obj;
    // Display the initial values of inherited member
    obj.modifyMember(10);
    // Display the modified values of inherited member
    obj.displayMember();
    return 0;
}
```

Output

```
Value of publicVar: 10
```

In the above example, we have accessed the public members of the base class in the derived class but we cannot access all the base class members directly in the derived class. It depends on the mode of inheritance and the access specifier in the base class.

Modes of Inheritance in C++:

Mode of inheritance controls the access level of the inherited members of the base class in the derived class. In C++, there are 3 modes of inheritance:

- **Public Mode**
- **Protected Mode**
- **Private Mode**

Public Inheritance Mode

If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

Example:

```
class ABC : public XYZ {...}            //  public derivation
```

Protected Inheritance Mode

If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

Example:

```
class ABC : protected XYZ {...}       //  protected derivation
```

Private Inheritance Mode

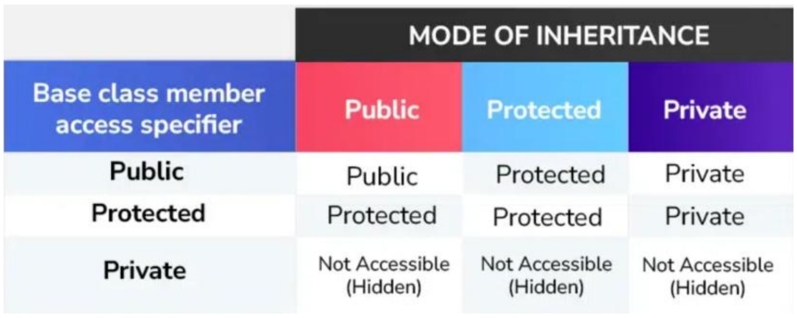
If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become private in the derived class. They can only be accessed by the member functions of the derived class.
Private mode is the default mode that is applied when we don't specify any mode.
Example:

```
class ABC : private XYZ {...}           //  private derivation
class ABC: XYZ {...}                    //  private derivation by default
```

Note: *The private members in the base class cannot be directly accessed in the derived class, while protected and public members can be directly accessed. To access or update the private members of the base class in derived class, we have to declare the derived class as friend class.*

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

Table 6.1Modes of Inheritance



	MODE OF INHERITANCE		
Base class member access specifier	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible (Hidden)	Not Accessible (Hidden)	Not Accessible (Hidden)

Examples of Modes of Inheritance

Example 1: Program to show different kinds of Inheritance Modes and their Member Access Levels

```
// C++ program to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
   int x;

protected:
   int y;
private:
   int z;
};
class B : public A {
   // x is public
   // y is protected
   // z is not accessible from B
};
class C : protected A {
   // x is protected
   // y is protected
   // z is not accessible from C
};
```

```
class D : private A // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

Example 2: Program to Access the Private Members of the Base Class in Derived Class

```
// C++ program to illustrate how to access the private data
// members of the base class in derived class using public
// getter methods of base class
#include <iostream>
using namespace std;

// Base class
class Base {
private:
   int privateVar;
public:
   // Constructor to initialize privateVar
   Base(int val): privateVar(val){}
   // Public getter function to get the value of privateVar
   int getPrivateVar() const { return privateVar; }
   // Public setter function to set the value of privateVar
   void setPrivateVar(int val) { privateVar = val; }
};
// Derived class
class Derived : public Base {
public:
   // Constructor to initialize the base class
   Derived(int val) : Base(val){}

   // Function to display the private member of the base
   // class
   void displayPrivateVar()
   {
      // Accessing privateVar using the public member
      // function of the base class
      cout << "Value of privateVar in Base class: "
           << getPrivateVar() << endl;
   }

   // Function to modify the private member of the base
   // class
   void modifyPrivateVar(int val)
   {
      // Modifying privateVar using the public member
      // function of the base class
      setPrivateVar(val);
   }
};

int main()
{
   // Create an object of Derived class
   Derived obj(10);

   // Display the initial value of privateVar
   obj.displayPrivateVar();

   // Modify the value of privateVar
   obj.modifyPrivateVar(20);

   // Display the modified value of privateVar
   obj.displayPrivateVar();

   return 0;
}
```

Output

```
Value of privateVar in Base class: 10

Value of privateVar in Base class: 20
```

The above program shows the method in which the private members of the base class remain encapsulated and are only accessible through controlled public or protected member functions.

We can also access the private members of the base class by declaring the derived class as friend class in the base class.

```
// C++ program to illustrate how to access the private
// members of the base class by declaring the derived class
// as friend class in the base class
#include <iostream>
using namespace std;

// Forward declaration of the Derived class
class Derived;

// Base class
class Base {
private:
    int privateVar;
public:
    // Constructor to initialize privateVar
    Base(int val)
        : privateVar(val)
    {
    }
    // Declare Derived class as a friend
    friend class Derived;
};
// Derived class
class Derived {
public:
   // Function to display the private member of the base
   // class
   void displayPrivateVar(Base& obj)
   {
      // Accessing privateVar directly since Derived is a
      // friend of Base
      cout << "Value of privateVar in Base class: "
          << obj.privateVar << endl;
   }
   // Function to modify the private member of the base
   // class
   void modifyPrivateVar(Base& obj, int val)
   {
      // Modifying privateVar directly since Derived is a
      // friend of Base
      obj.privateVar = val;
   }
};
int main()
{
   // Create an object of Base class
   Base baseObj(10);
   // Create an object of Derived class
   Derived derivedObj;
   // Display the initial value of privateVar
   derivedObj.displayPrivateVar(baseObj);
   // Modify the value of privateVar
   derivedObj.modifyPrivateVar(baseObj, 20);

   // Display the modified value of privateVar
   derivedObj.displayPrivateVar(baseObj);
   return 0;
}
```

Output

```
Value of privateVar in Base class: 10

Value of privateVar in Base class: 20
```

Types Of Inheritance in C++:

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have 5 types of inheritances:

1. **Single inheritance**
2. **Multilevel inheritance**

3. **Multiple inheritance**
4. **Hierarchical inheritance**
5. **Hybrid inheritance**

1. Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.

Syntax

```
class subclass_name : access_mode base_class
{
  //  body of subclass
};
```

Example:

```
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
```

Implementation:

```
// C++ program to demonstrate how to implement the Single
// inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
// sub class derived from a single base classes
class Car : public Vehicle {
public:
    Car() { cout << "This Vehicle is Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle

This Vehicle is Car
```

2. Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.

Syntax

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified and can be different.

Example:

```
class A
{
... .. ...
};
class B
{
... .. ...
};
class C: public A, public B
{
... ... ...
};
```

Implementation:

```
// C++ program to illustrate the multiple inheritance
#include <iostream>
using namespace std;
// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
// second base class
class FourWheeler {
public:
    FourWheeler() { cout << "This is a 4 Wheeler\n"; }
};
// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
  public:
    Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle

This is a 4 Wheeler

This 4 Wheeler Vehical is a Car
```

3. Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels.

Syntax

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier derived_class1
{
... .. ...
}
.....
```

Example:

```
class C
{
... .. ...
};
class B : public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

Implementation

```
// C++ program to implement Multilevel Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
   fourWheeler() { cout << "4 Wheeler Vehicles\n"; }
};
// sub class derived from the derived base class fourWheeler
```

```
class Car : public fourWheeler {
public:
   Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

Output

```
This is a Vehicle

4 Wheeler Vehicles

This 4 Wheeler Vehical is a Car
```

4. Hierarchical Inheritance

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

Syntax

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier base_class
{
... .. ...
}
```

Example:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Implementation

```
// C++ program to implement Hierarchical Inheritance
```

```cpp
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// first sub class
class Car : public Vehicle {
public:
   Car() { cout << "This Vehicle is Car\n"; }
};
// second sub class
class Bus : public Vehicle {
public:
   Bus() { cout << "This Vehicle is Bus\n"; }
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output

```
This is a Vehicle

This Vehicle is Car

This is a Vehicle

This Vehicle is Bus
```

5. Hybrid Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++

There is no particular syntax of hybrid inheritance. We can just combine two of the above inheritance types.

Example:

```
class F
{
... .. ...
}
class G
{
... .. ...
}
```

```
class B : public F
{
… .. …
}
class E : public F, public G
{
… .. …
}
class A : public B {
… .. …
}
class C : public B {
… .. …
}
```

Implementation:

```
// C++ program to illustrate the implementation of Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
  public:
  Car() { cout << "This Vehical is a Car\n"; }
};

// second sub class
class Bus : public Vehicle, public Fare {
  public:
  Bus() { cout << "This Vehicle is a Bus with Fare\n"; }
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Bus obj2;
   return 0;
}
```

Output

```
This is a Vehicle

Fare of Vehicle

This Vehicle is a Bus with Fare
```

A Special Case of Hybrid Inheritance: Multipath Inheritance
In multipath inheritance, a class is derived from two base classes and these two base classes in turn are derived from one common base class. An ambiguity can arise in this type of inheritance in the most derived class. This problem is also called diamond problem due to shape of inherited path.

Constructors and Destructors in Inheritance:

Constructors and Destructors are generally defined by the programmer and if not, the compiler automatically creates them, so they are present in every class in C++. Now, the question arises what happens to the constructor and destructor when a class is inherited by another class.
In C++ inheritance, the **constructors and destructors are not inherited by the derived class,** but we can call the constructor of the base class in derived class.

- The constructors will be called by the complier in the order in which they are inherited. It means that base class constructors will be called first, then derived class constructors will be called.
- The destructors will be called in reverse order in which the compiler is declared.
- We can also call the constructors and destructors manually in the derived class.

Example

```
// C++ program to show the order of constructor call
// in single inheritance

#include <iostream>
using namespace std;
// base class
class Parent {
public:
   // base class constructor
   Parent() { cout << "Inside base class" << endl; }
};

// sub class
class Child : public Parent {
public:
   // sub class constructor
   Child() { cout << "Inside sub class" << endl; }
};

// main function
int main()
{
   // creating object of sub class
   Child obj;
   return 0;
}
```

Output

```
Inside base class

Inside sub class
```

7. POLYMORPHISM

C++ 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 real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So, the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.
Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**

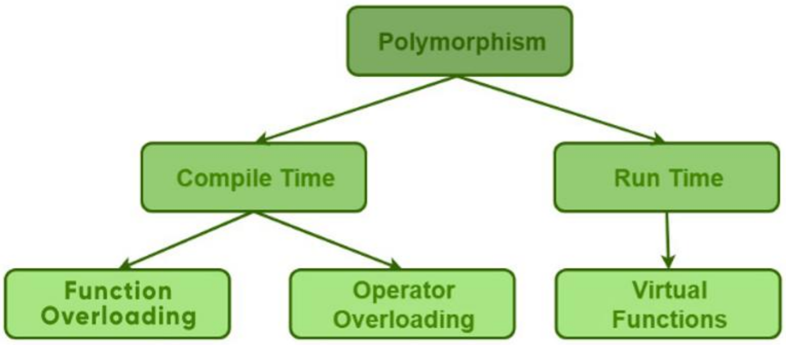


Fig 7.1 Types of Polymorphism

1. **Compile-Time Polymorphism**
This type of polymorphism is achieved by function overloading or operator overloading. Function overloading and operator overloading is already discussed.
2. **Runtime Polymorphism**
This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Runtime Polymorphism with Data Members
Runtime Polymorphism cannot be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable of parent class which refers to the instance of the derived class.

```
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

//  base class declaration.
```

```
class Animal {
public:
   string color = "Black";
};
// inheriting Animal class.
class Dog : public Animal {
public:
   string color = "Grey";
};
// Driver code
int main(void)
{
   Animal d = Dog(); // accessing the field by reference
                // variable which refers to derived
   cout << d.color;
}
```

Output

```
Black
```

We can see that the parent class reference will always refer to the data member of the parent class.

B. Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Below is the C++ program to demonstrate virtual function:

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
   // virtual function
   virtual void display()
   {
      cout << "Called virtual Base Class function"
          << "\n\n";
   }
   void print()
   {
      cout << "Called GFG_Base print function"
          << "\n\n";
```

```
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {
public:
    void display()
    {
        cout << "Called GFG_Child Display Function"
             << "\n\n";
    }
    void print()
    {
        cout << "Called GFG_Child print Function"
             << "\n\n";
    }
};

int main()
{
    // Create a reference of class GFG_Base
    GFG_Base* base;

    GFG_Child child;

    base = &child;
    // This will call the virtual function
    base->display();
    // This will call the non-virtual function
    base->print();
}
```

Output

```
Called GFG_Child Display Function


Called GFG_Base print function
```

Example 2:

```
// C++ program for virtual function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }
    void show() { cout << "show base class" << endl; }
};
class derived : public base {
public:
    // print () is already virtual function in
    // derived class, we could also declared as
    // virtual void print () explicitly
    void print() { cout << "print derived class" << endl; }
    void show() { cout << "show derived class" << endl; }
};
// Driver code
int main()
{
    base* bptr;
    derived d;
    bptr = &d;
    // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();
    // Non-virtual function, binded
    // at compile time
    bptr->show();

    return 0;
}
```

Output

```
print derived class

show base class
```

Pure Virtual Functions and Abstract Classes in C++:

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**. For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw(). Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
A **pure virtual function** (or abstract function) in C++ is a virtual function for which we can have an implementation, but we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declaration.

Example of Pure Virtual Functions

```
// An abstract class
class Test {
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Complete Example

A pure virtual function is implemented by classes that are derived from an Abstract class.

```
// C++ Program to illustrate the abstract class and virtual
// functions
#include <iostream>
using namespace std;
class Base {
    // private member variable
    int x;
public:
    // pure virtual function
    virtual void fun() = 0;
    // getter function to access x
    int getX() { return x; }
};
// This class inherits from Base and implements fun()
class Derived : public Base {
    // private member variable
    int y;

public:
    // implementation of the pure virtual function
    void fun() { cout << "fun() called"; }
};
int main(void)
{
    // creating an object of Derived class
    Derived d;

    // calling the fun() function of Derived class
    d.fun();
    return 0;
}
```

Output

```
fun() called
```

Some Interesting Facts

1. A class is abstract if it has at least one pure virtual function.

Example

In the below C++ code, Test is an abstract class because it has a pure virtual function show().

```
// C++ program to illustrate the abstract class with pure
// virtual functions
```

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

class Test {
   // private member variable
   int x;
public:
   // pure virtual function
   virtual void show() = 0;
   // getter function to access x
   int getX() { return x; }
};
int main(void)
{
   // Error: Cannot instantiate an abstract class
   Test t;
   return 0;
}
```

Output

```
Compiler Error: cannot declare variable 't' to be of abstract
 type 'Test' because the following virtual functions are pure
within 'Test': note:     virtual void Test::show()
```

2. We can have pointers and references of abstract class type.

For example, the following program works fine.

```
// C++ program that demonstrate that
// we can have pointers and references
// of abstract class type.

#include <iostream>
using namespace std;
class Base {
public:
   // pure virtual function
   virtual void show() = 0;
};
class Derived : public Base {
public:
   // implementation of the pure virtual function
   void show() { cout << "In Derived \n"; }
};
int main(void)
{
   // creating a pointer of type
   // Base pointing to an object
   // of type Derived
   Base* bp = new Derived();
   // calling the show() function using the
   // pointer
   bp->show();
```

```
    return 0;
}
```

Output

```
In Derived
```

3. If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.

The following example demonstrates the same.

```
// C++ program to demonstrate that if we do not override
// the pure virtual function in the derived class, then
// the derived class also becomes an abstract class

#include <iostream>
using namespace std;

class Base {
public:
    // pure virtual function
    virtual void show() = 0;
};

class Derived : public Base {
};

int main(void)
{
    // creating an object of Derived class
    Derived d;
    return 0;
}
```

Output

```
Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived'  because the following virtual functions are pure within
'Derived': virtual void Base::show()
```

4. An abstract class can have constructors.

For example, the following program compiles and runs fine.

```
// C++ program to demonstrate that
// an abstract class can have constructors.

#include <iostream>
using namespace std;

// An abstract class with constructor
class Base {
protected:
    // protected member variable
    int x;
public:
    // pure virtual function
    virtual void fun() = 0;
    // constructor of Base class
    Base(int i)
    {
        x = i;
        cout << "Constructor of base called\n";
    }
};
class Derived : public Base {
    // private member variable
    int y;
public:
    // calling the constructor of Base class
    Derived(int i, int j)
        : Base(i)
    {
        y = j;
    }

    // implementation of pure virtual function
    void fun()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main(void)
{
    // creating an object of Derived class
    Derived d(4, 5);

    // calling the fun() function of Derived class
    d.fun();

    // creating an object of Derived class using
    // a pointer of the Base class
    Base* ptr = new Derived(6, 7);

    // calling the fun() function using the
    // pointer
    ptr->fun();

    return 0;
}
```

Output

```
Constructor of base called
```

```
x = 4, y = 5

Constructor of base called

x = 6, y = 7
```

5. An abstract class in C++ can also be defined using struct keyword.
Example

```
struct shapeClass
{
   virtual void Draw()=0;
}
```

Virtual Destructor:

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor result in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
For example, the following program results in undefined behavior.

```
// CPP program without virtual destructor

// causing undefined behavior

#include <iostream>

using namespace std;

class base {

  public:

    base()

    { cout << "Constructing base\n"; }

    ~base()

    { cout<< "Destructing base\n"; }

};

class derived: public base {

  public:

    derived()

     { cout << "Constructing derived\n"; }

    ~derived()

       { cout << "Destructing derived\n"; }

};
```

```
int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
// A program with virtual destructor
#include <iostream>
using namespace std;
class base {
  public:
    base()
    { cout << "Constructing base\n"; }
    virtual ~base()
    { cout << "Destructing base\n"; }
};
class derived : public base {
  public:
    derived()
    { cout << "Constructing derived\n"; }
    ~derived()
    { cout << "Destructing derived\n"; }
};
int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

8. FILE AND STREAMS

File Handling through C++ Classes:

File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk). How to achieve the File Handling?
For achieving file handling we need to follow the following steps:-
STEP 1-Naming a file
STEP 2-Opening a file
STEP 3-Writing data into the file
STEP 4-Reading data from the file
STEP 5-Closing a file.

Streams in C++ :-

We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream. In other words, streams are nothing but the flow of data in a sequence.
The input and output operation between the executing program and the devices like keyboard and monitor are known as "console I/O operation". The input and output operation between the executing program and files are known as "disk I/O operation".

Classes for File stream operations:-

The I/O system of C++ contains a set of classes which define the file handling methods. These include ifstream, ofstream and fstream classes. These classes are derived from fstream and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files. File handling is essential for data storage and retrieval in applications.

1. ios:-
   - ios stands for input output stream.
   - This class is the base class for other classes in this class hierarchy.
   - This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-
   - istream stands for input stream.
   - This class is derived from the class 'ios'.
   - This class handle input stream.
   - The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
   - This class declares input functions such as get(), getline() and read().

3. ostream:-
   - ostream stands for output stream.
   - This class is derived from the class 'ios'.
   - This class handle output stream.
   - The insertion operator (<<) is overloaded in this class to handle output streams to files from the program execution.
   - This class declares output functions such as put() and write().

4. streambuf:-
   - This class contains a pointer which points to the buffer which is used to manage the input and output streams.

5. fstreambase:-
- This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
- This class contains open() and close() function.

6. ifstream:-
- This class provides input operations.
- It contains open() function with default input mode.
- Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

7. ofstream:-
- This class provides output operations.
- It contains open() function with default output mode.
- Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

8. fstream:-
- This class provides support for simultaneous input and output operations.
- Inherits all the functions from istream and ostream classes through iostream.

9. filebuf:-
- Its purpose is to set the file buffers to read and write.
- We can also use file buffer member function to determine the length of the file.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.

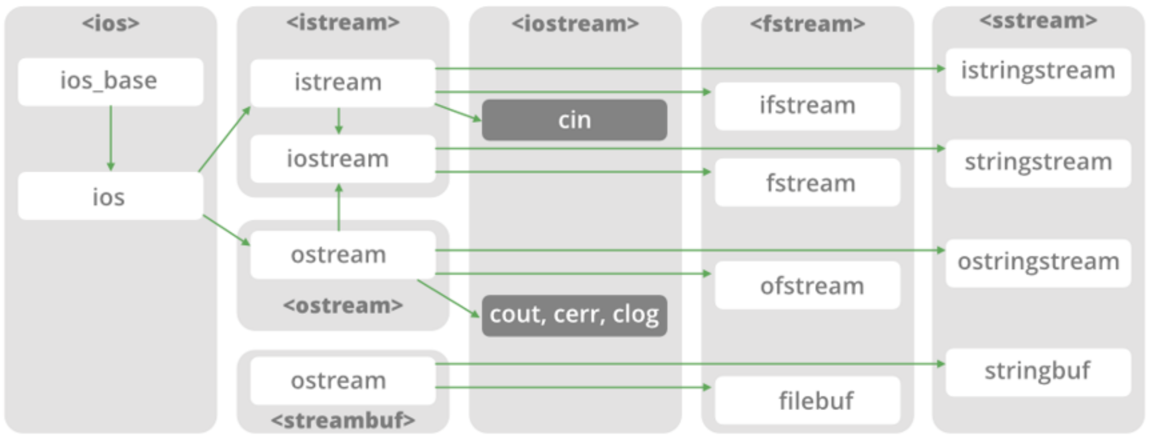


Fig 8.1 header files and their classes

Now the first step to open the particular file for read or write operation. We can open file by
1. passing file name in constructor at the time of object creation
2. using the open method
For e.g.
Open File by using constructor
ifstream (const char filename, ios_base::openmode mode = ios_base::in);*
ifstream fin(filename, openmode) //by default openmode = ios::in
ifstream fin("filename");
Open File by using open method
//Calling of default constructor

ifstream fin;
fin.open(filename, openmode)
fin.open("filename");

Table 8.1 file opening modes

Member Constant	**Stands For**	**Access**
ios::in	input	File open for reading: the internal stream buffer supports input operations.
ios::out	output	File open for writing: the internal stream buffer supports output operations.
ios::binary	binary	Operations are performed in binary mode rather than text.
ios::ate	at end	The output position starts at the end of the file.
ios::app	append	All output operations happen at the end of the file, appending to its existing contents.
ios::trunc	truncate	Any contents that existed in the file before it is open are discarded.
ios::nocreate	Do not create	Does not allow to create new file if it does not exist.
ios::noreplace	Do not replace	Does not replace old file with new file.

Both ios::app and ios::ate take us to the end of the file when it is opened. The difference between the two modes is that ios :: app allow us to add data to the end of the file only, while ios :: ate mode permits us add data or to modify the existing data anywhere in the file.

Table 8.2 Default Open Modes

ifstream	ios::in
ofstream	ios::out
fstream	ios::in | ios::out

Problem Statement : To read and write a File in C++.
Examples:

```
Input :
Welcome, we live in a beautiful world!
-1
Output :
Welcome, we live in a beautiful world!
```

Below is the implementation by using **ifstream & ofstream classes**.

```
/* File Handling with C++ using ifstream & ofstream class object*/
/* To write the Content in File*/
/* Then to read the content of file*/
#include <iostream>

/* fstream header file for ifstream, ofstream,
  fstream classes */
#include <fstream>

using namespace std;

// Driver Code
int main()
{
    // Creation of ofstream class object
    ofstream fout;

    string line;

    // by default ios::out mode, automatically deletes
    // the content of file. To append the content, open in ios:app
    // fout.open("sample.txt", ios::app)
    fout.open("sample.txt");

    // Execute a loop If file successfully opened
    while (fout) {

        // Read a Line from standard input
        getline(cin, line);

        // Press -1 to exit
        if (line == "-1")
            break;

        // Write line in file
        fout << line << endl;
    }

    // Close the File
    fout.close();

    // Creation of ifstream class object to read the file
    ifstream fin;

    // by default open mode = ios::in mode
    fin.open("sample.txt");

    // Execute a loop until EOF (End of File)
    while (getline(fin, line)) {

        // Print line (read from file) in Console
        cout << line << endl;
    }

    // Close the file
    fin.close();

    return 0;
}
```

Below is the implementation by using **fstream class**.

```
/* File Handling with C++ using fstream class object */
/* To write the Content in File */
/* Then to read the content of file*/
#include <iostream>

/* fstream header file for ifstream, ofstream,
   fstream classes */
#include <fstream>

using namespace std;

// Driver Code
int main()
{
    // Creation of fstream class object
    fstream fio;

    string line;

    // by default openmode = ios::in|ios::out mode
    // Automatically overwrites the content of file, To append
    // the content, open in ios:app
    // fio.open("sample.txt", ios::in|ios::out|ios::app)
    // ios::trunc mode delete all content before open
    fio.open("sample.txt", ios::trunc | ios::out | ios::in);

    // Execute a loop If file successfully Opened
    while (fio) {

       // Read a Line from standard input
       getline(cin, line);

       // Press -1 to exit
       if (line == "-1")
          break;

       // Write line in file
       fio << line << endl;
    }

    // Execute a loop until EOF (End of File)
    // point read pointer at beginning of file
    fio.seekg(0, ios::beg);

    while (fio) {

       // Read a Line from File
       getline(fio, line);

       // Print line in Console
       cout << line << endl;
    }

    // Close the file
    fio.close();

    return 0;
}
```

Q: write a single file handling program in c++ to reading and writing data on a file.

```
// Include necessary header files
#include <fstream>
#include <iostream>
#include <string>

// Use the standard namespace
using namespace std;
```

```
int main()
{
   // Create an output file stream object
   ofstream fout;
   // Open a file named "NewFile.txt" for writing
   fout.open("NewFile.txt");

   // Check if the file opened successfully
   if (!fout) {
      // Print an error message if the file couldn't be
      // opened
      cerr << "Error opening file!" << endl;
      // Return 1 to indicate failure
      return 1;
   }

   // Declare a string variable to hold each line of text
   string line;
   // Initialize a counter to limit input to 5 lines
   int i = 0;

   // Prompt the user to enter 5 lines of text
   cout << "Enter 5 lines of text:" << endl;
   // Loop to read 5 lines of input from the user
   while (i < 5) {
      // Read a line of text from standard input
      getline(cin, line);
      // Write the line of text to the file
      fout << line << endl;
      // Increment the counter
      i += 1;
   }

   // Close the file after writing
   fout.close();

   // Print a success message
   cout << "Text successfully written to NewFile.txt"
        << endl;

   // Return 0 to indicate successful execution
   return 0;
}
```

Output

```
Enter 5 lines of text:
Hi, Welcome Everyone
Learn to code
C++
Java
Python
>>Text successfully written to NewFile.txt
```

Q: WA C++ file handling program to read data from the file called student.doc

```cpp
#include<iostream>
#include<fstream>

using namespace std;

main()
{
    int rno,fee;
    char name[50];

    ifstream fin("d:/student.doc");

    fin>>rno>>name>>fee;   //read data from the file student

    fin.close();

    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

    return 0;
}
```