MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF

COMPUTER APPLICATION

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING CONCEPTS USING C++

SUBJECT CODE: SU22A

SEMESTER: I

PREPARED BY: PROF. B.HEMALATHA

Syllabus

CORE - II OBJECT ORIENTED PROGRAMMING CONCEPTS USING C++

I YEAR / II SEM

OBJECTIVES: To inculcate knowledge on Object-oriented programming concepts using C++.

- To gain Knowledge on programming with C++.

- OUTCOMES: To write programs using OOP concepts like Abstraction, Encapsulation, Inheritance and Polymorphism

- UNIT - I Introduction to C++ - key concepts of Object-Oriented Programming –Advantages – Object Oriented Languages – I/O in C++ - C++ Declarations. Control Structures : - Decision Making and Statements : If ..else, jump, goto, break, continue, Switch case statements - Loops in C++ : for, while, do - functions in C++ - inline functions – Function Overloading.

UNIT - II Classes and Objects: Declaring Objects – Defining Member Functions – Static Member variables and functions – array of objects –friend functions – Overloading member functions – Bit fields and classes – Constructor and destructor with static members.

UNIT- III Operator Overloading: Overloading unary, binary operators – Overloading Friend functions – type conversion – Inheritance: Types of Inheritance – Single, Multilevel, Multiple, Hierarchal, Hybrid, Multi path inheritance – Virtual base Classes – Abstract Classes.

UNIT - IV Pointers – Declaration – Pointer to Class , Object – this pointer – Pointers to derived classes and Base classes – Arrays – Characteristics – array of classes – Memory models – new and delete operators – dynamic object – Binding, Polymorphism and Virtual Functions.

UNIT - V Files – File stream classes – file modes – Sequential Read / Write operations – Binary and ASCII Files – Random Access Operation – Templates – Exception Handling - String – Declaring and Initializing string objects – String Attributes – Miscellaneous functions .

UNIT I

❖ Introduction to C++

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features. C++ is a statically typed, free form, multiparadigm, compiled general-purpose language.

C++ is an **Object Oriented Programming language** but is not purely Object Oriented. Its features like Friend and Virtual, violate some of the very important OOPS features, rendering this language unworthy of being called completely Object Oriented. Its a middle level language.

Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C is a procedural language. Apart form this there are many other features of C++ which gives this language an upper hand on C laguage.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.

7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

Syntax and Structure of C++ program

Here we will discuss one simple and basic C++ program to print "Hello this is C++" and its structure in parts with details and uses.

```
#include <iostream.h>
using namespace std;
int main()
{
   cout << "Hello this is C++";
}
```

Header files are included at the beginning just like in C program. Here iostream is a header file which provides us with input & output streams. Header files contained predeclared function libraries, which can be used by users for their ease.

Using namespace std, tells the compiler to use standard namespace. Namespace collects identifiers used for class, object and variables. NameSpace can be used by two ways in a program, either by the use of using statement at the beginning, like we did in above mentioned program or by using name of namespace as prefix before the identifier with scope resolution (::) operator.

Example: std::cout << "A";

main(), is the function which holds the executing part of program its return type is int.

cout <<, is used to print anything on screen, same as printf in C language. **cin** and **cout** are same as scanf and printf, only difference is that you do not need to mention format specifiers like, %d for int etc, in cout & cin.

Comments in C++ Program

For single line comments, use // before mentioning comment, like

```
cout<<"single line";   // This is single line comment
```

For multiple line comment, enclose the comment between **/*** and ***/**

```
/*this is
  a multiple line
  comment */
```

Variables can be declared anywhere in the entire program, but must be declared, before they are used. Hence, we don't need to declare variable at the start of the program.

Datatypes and Modifiers in C++

They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be of two types:

1. Built-in Datatypes
2. User-defined or Abstract Datatypes

Built-in Data Types

These are the datatypes which are predefined and are wired directly into the compiler. For eg: int, char etc.

User defined or Abstract data types

These are the type, that user creates as a class or a structure. In C++ these are classes where as in C language user-defined datatypes were implemented as structures.

Basic Built in Datatypes in C++

char	for character storage (1 byte)
int	for integral number (2 bytes)
float	single precision floating point (4 bytes)
double	double precision floating point numbers (8 bytes)

Example:

```
char a = 'A';          // character type
int a = 1;             // integer type
float a = 3.14159;     // floating point type
double a = 6e-4;       // double type (e is for exponential)
```

Other Built in Datatypes in C++

bool	Boolean (True or False)
void	Without any Value
wchar_t	Wide Character

Enum as Datatype in C++: Enumerated type declares a new type-name along with a sequence of values containing identifiers which has values starting from 0 and incrementing by 1 every time.

For Example: enum day(mon, tues, wed, thurs, fri) d; Here an enumeration of days is defined which is represented by the variable d. **mon** will hold value **0**, **tue** will have **1** and so on. We can also explicitly assign values, like, enum day(mon, tue=7, wed);. Here, **mon** will be **0**, **tue** will be assigned **7**, so **wed** will get value **8**.

Modifiers in C++

In C++, special words(called **modifiers**) can be used to modify the meaning of the predefined built-in data types and expand them to a much larger set. There are four datatype modifiers in C++, they are:

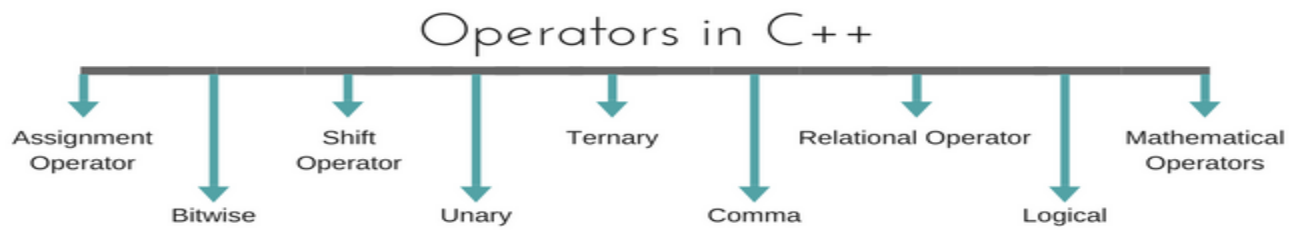
1. long
2. short
3. signed
4. unsigned

The above mentioned modifiers can be used along with built in datatypes to make them more precise and even expand their range.

Below mentioned are some important points you must know about the modifiers,

1. **long** and **short** modify the maximum and minimum values that a data type will hold.
2. A plain int must have a minimum size of **short**.
3. Size hierarchy : short int < int < long int
4. Size hierarchy for floating point numbers is : float < double < long double
5. **long float** is not a legal type and there are no **short floating point** numbers.
6. **Signed** types includes both positive and negative numbers and is the default type.

Unsigned**, numbers are always without any sign, that is always positive.** Operators in C++ Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

- Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

- Arithmetic or Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

C++ and C also use a shorthand notation to perform an operation and assignment at same type. *Example*,

```
int  x=10;
x += 4 // will add 4 to 10, and hence assign 14 to X.
x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

- Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any Variables, whereas equivalent operator is used to compare values, like in if-else conditions, *Example*

```
int x = 10;  //assignment operator
x=5;        // again assignment operator
if(x == 5)   // here we have used equivalent relational operator, for comparison
{
   cout <<"Successfully compared";
}
```

- Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

- Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like char, int and long and not with floating point values.

- Bitwise AND operators &
- Bitwise OR operator |
- And bitwise XOR operator ^
- And, bitwise NOT operator ~

They can be used as shorthand notation too, & = , |= , ^= , ~= etc.

- Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

- Left Shift Operator <<
- Right Shift Operator >>
- Unsigned Right Shift Operator >>>

- Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment ++ and decrement -- operators are most used. Other Unary Operators : address of &, dereference *, new and delete, bitwise not ~, logical not !, unary minus - and unary plus +.

- Ternary Operator

The ternary if-else ? : is an operator which has three operands.

```
int a = 10;
a > 5 ? cout << "true" : cout << "false"
```

- Comma Operator

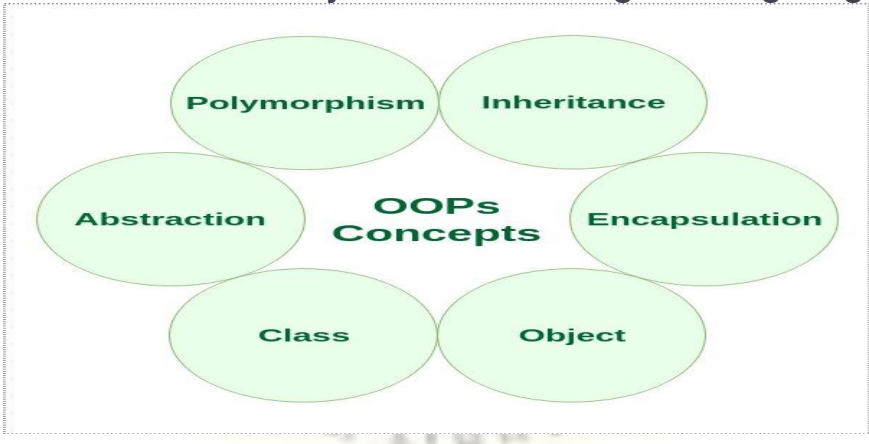
This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used. *Example* :

```
int a,b,c; // variables declaration using comma operator

a=b++, c++; // a = c++ will be done.
```

❖ Key Concepts of object-oriented programming

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

Characteristics of an Object Oriented Programming language



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

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

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

- ❖ **Object**: An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

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

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

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

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

- ❖ **Encapsulation**:

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like

the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

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

- **Abstraction**

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm

according to which the function is actually calculating the power of numbers.

❖ **Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

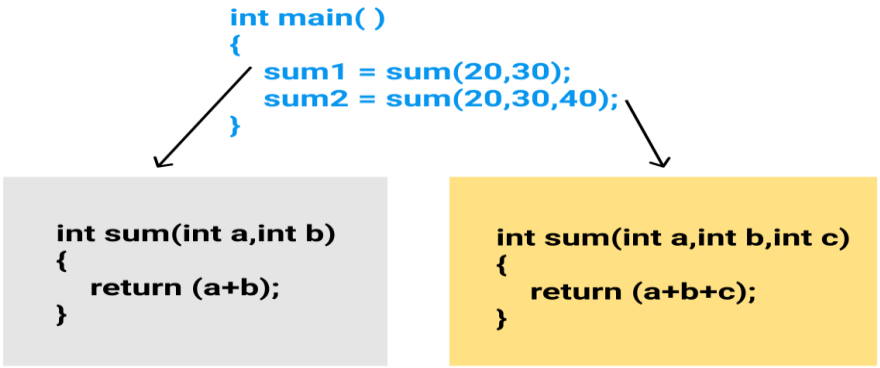
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

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

C++ supports operator overloading and function overloading.

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

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

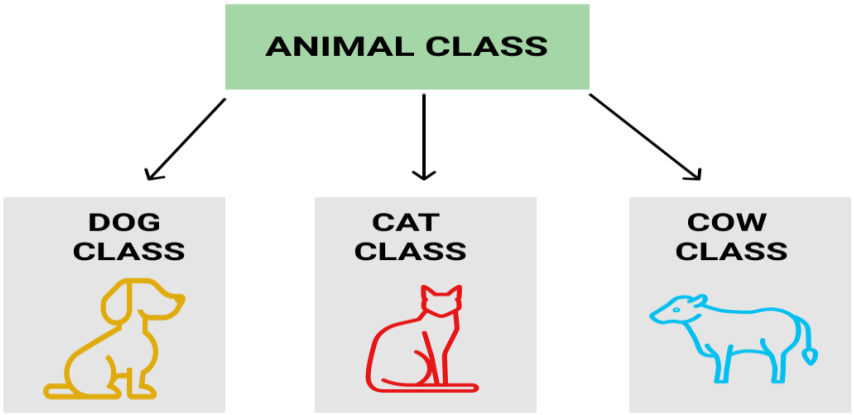


❖ **Inheritance:** The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of

Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**: The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

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



- **Dynamic Binding:** In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.
- **Message Passing:** Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Advantages of object oriented languages.

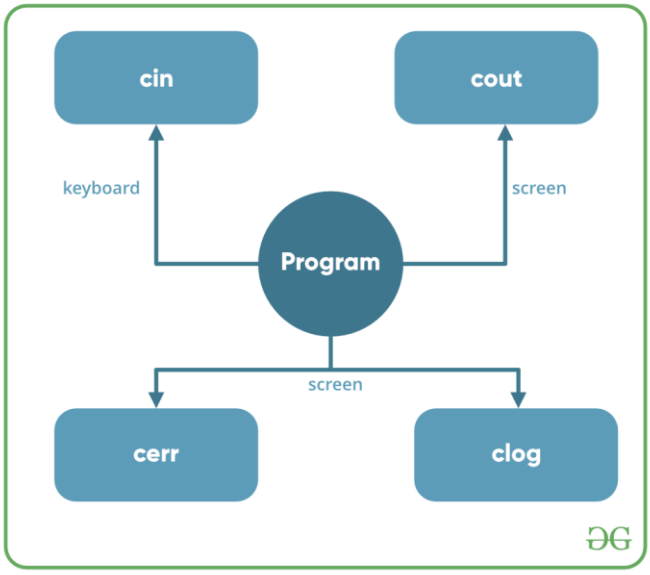
- OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

- OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
- OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

❖ I/O in C++

C++ comes with libraries that provide us with many ways for performing input and output. 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.



Header files available in C++ for Input/Output operations are:

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.
2. **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.

3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two keywords **cout in C++** and **cin in C++** 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++. To use cin and cout in C++ one must include the header file *iostream* in the program.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is 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(<<).

```
#include <iostream>
using namespace std;
int main()
{
   char sample[] = "GeeksforGeeks";
   cout << sample << " - A computer science portal for geeks";
   return 0;
}
```

Output:
GeeksforGeeks - A computer science portal for geeks

In the above program the insertion operator(<<) inserts the value of the string variable **sample** followed by the string “A computer science portal for geeks” in the standard output stream **cout** which is then displayed on screen.

- **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **iostream** 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 reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

```
#include <iostream.h>
int main()
{
   int age;

   cout << "Enter your age:";
   cin >> age;
   cout << "\nYour age is: " << age;

   return 0;
}
```

Input :
18
Output:
Enter your age:
Your age is: 18

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator (>>) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

❖ C++ Declarations

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive .

There are following basic types of variable in C++

S.No	Type & Description
1	bool Stores either value true or false.
2	char Typically a single octet (one byte). This is an integer type.
3	int The most natural size of integer for the machine.
4	float A single-precision floating point value.
5	double A double-precision floating point value.
6	void Represents the absence of type.
7	wchar_t A wide character type.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
  // Variable definition:
  int a, b;
  int c;
  float f;

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

  cout << c << endl ;

  f = 70.0/3.0;
  cout << f << endl ;

  return 0;
}
```

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

```
30
23.3333
```

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

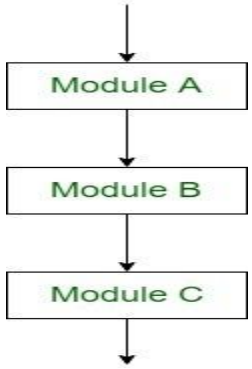
❖ Control structures

Control structures are used to alter the flow of execution of the program. Why do we need to alter the program flow? The reason is "**decision making**"! In life, we may be given with a set of option like doing "Electronics" or "Computer science". We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc). With the decision we make, we alter the flow of our life's direction. This is exactly what happens in a C++ program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.

There are **three types** of control structures available in C++

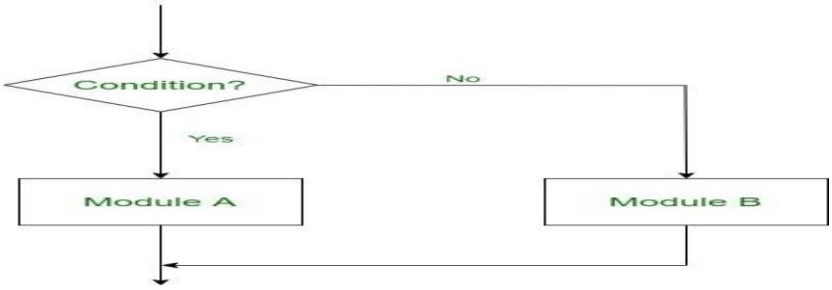
1) Sequence structure (straight line paths)

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



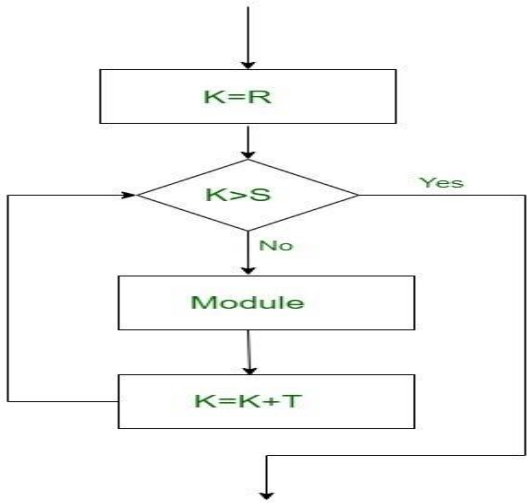
Sequential Control flow

2) Selection structure (one or many branches): Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as **Conditional Structures**.



3) Loop structure (repetition of a set of activities)

The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.



❖ Decision making and statements

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

Sr.No	**Statement & Description**
1	if statement An 'if' statement consists of a boolean expression followed by one or more statements.
2	if...else statement An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	switch statement A 'switch' statement allows a variable to be tested for equality against a list of values.
4	nested if statements You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	nested switch statements You can use one 'switch' statement inside another 'switch' statement(s).

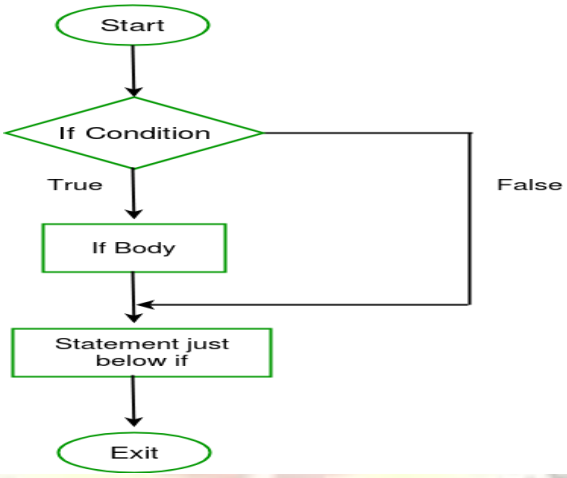
if statement in C++

if statement is the most simple 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 statement is executed otherwise not.

Syntax:

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

Flowchart




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

 int main()
{
    int i = 10;

    if (i > 15)
    {
      cout<<"10 is less than 15";
    }

    cout<<"I am Not in if";
 }
```

Output:

```
I am Not in if
```

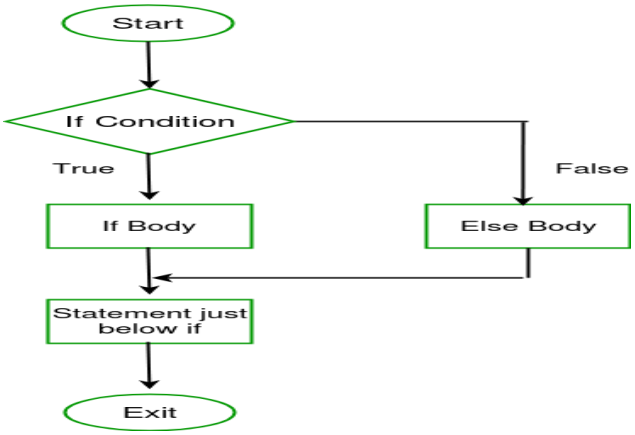
if-else statement

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 if the condition is false. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

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

Flowchart:



Example:

```
// C++ program to illustrate if-else statement
#include<iostream>
using namespace std;

int main()
 {
      int i = 20;

      if (i < 15)
         cout<<"i is smaller than 15";
      else
         cout<<"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.

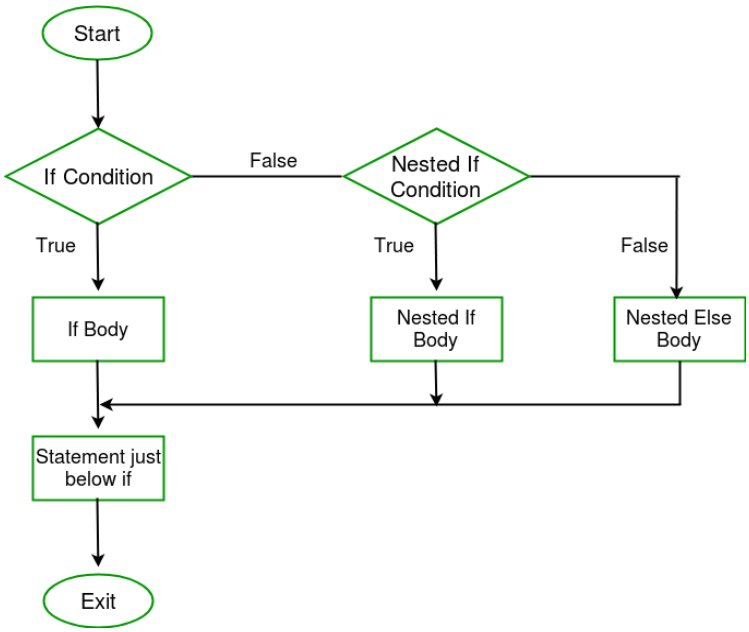
nested-if statement

Nested if statements means an if statement inside another if statement. Yes, C++ allows us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Flowchart



Example:

```
// C++ program to illustrate nested-if statement
#include <iostream>
using namespace std;

int main()
{
   int i = 10;

   if (i == 10)
   {
      // First if statement
      if (i < 15)
         cout<<"i is smaller than 15\n";

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

   return 0;
}
```

Output:
i is smaller than 15
i is smaller than 12 too

if-else-if ladder

Here, a user can decide among multiple options. The 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++ else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

Syntax:

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


```
// C++ program to illustrate if-else-if ladder
#include<iostream>
using namespace std;

int main()
{
   int i = 20;
```

```
    if (i == 10)
        cout<<"i is 10";
    else if (i == 15)
        cout<<"i is 15";
    else if (i == 20)
        cout<<"i is 20";
    else
        cout<<"i is not present";
}
```

Output:
i is 20

❖ Goto , break, continue, Switch case statements

Jump Statements in C++

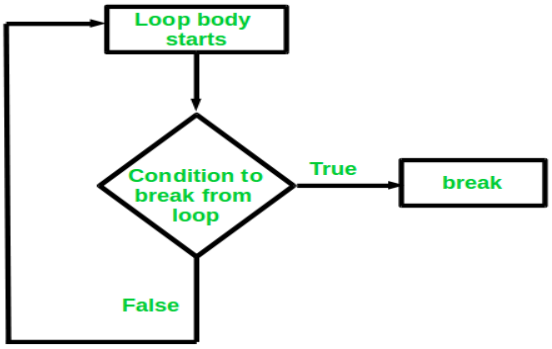
These statements are used in C++ for unconditional flow of control through out the funtions in a program. They support four type of jump statements:

1. **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 stops there and control returns from the loop immediately to the first statement after the loop.

   Syntax:

   ```
   break;
   ```

   Basically break statements are used in the 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.



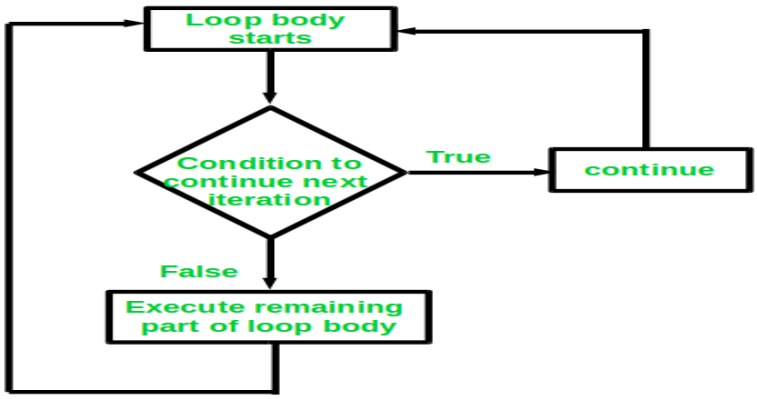
Example:

```
// CPP program to illustrate
// Linear Search
#include <iostream>
using namespace std;

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) {
         cout << "Element found at position: " << (i + 1);
         break;
      }
   }
}

// Driver program to test above function
int main()
{
   int arr[] = { 1, 2, 3, 4, 5, 6 };
   int n = 6; // no of elements
   int key = 3; // key to be searched

   // Calling function to find the key
   findElement(arr, n, key);

   return 0;
}
```

Output:
Element found at position: 3

2.**Continue statement**: Continue is also a loop control statement just like the break statement. *continue* statement is opposite to that of break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggest 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 next iteration of the loop will begin.
Syntax:

```
continue;
```




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

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
         cout << i << " ";
   }

   return 0;
}
```

Output:

1 2 3 4 5 7 8 9 10

The *continue* statement can be used with any other loop also like while or do while in a similar way as it is used with for loop above

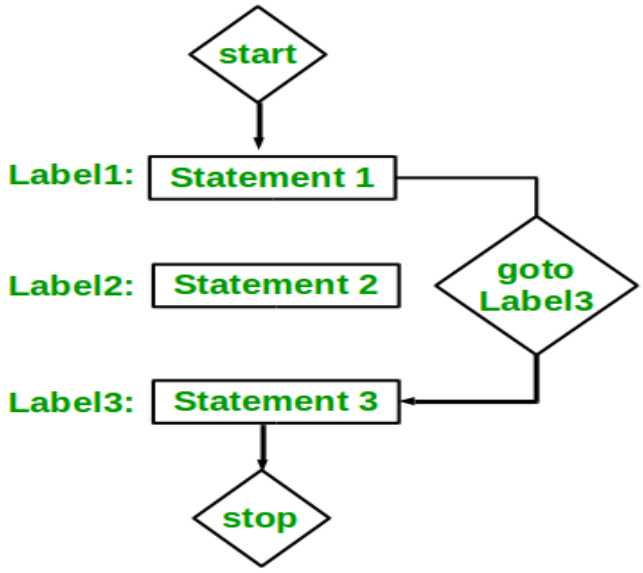
3.**goto statement**

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

Syntax:

```
Syntax1       |   Syntax2
-----------------------------
goto label;   |    label:
.             |    .
.             |    .
.             |    .
label:        |   goto label;
```

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier which indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



Below are some examples on how to use goto statement:

Example:

```
// C++ program to check if a number is
// even or not using goto statement
#include <iostream>
using namespace std;

// function to check even or not
void checkEvenOrNot(int num)
{
   if (num % 2 == 0)
   // jump to even
      goto even;
   else
   // jump to odd
      goto odd;

even:
   cout << num << " is even";
   // return if even
   return;
odd:
   cout << num << " is odd";
}

// Driver program to test above function
int main()
{
   int num = 26;
   checkEvenOrNot(num);
   return 0;
}
```

Disadvantages of using goto statement:

- The use of goto statement is highly discouraged as it makes the program logic very complex.
- use of goto makes the task of analyzing and verifying the correctness of programs (particularly those involving loops) very difficult.
- Use of goto can be simply avoided using break and continue statements.

4.Switch Statement

Switch case statements are a substitute for long if statements that compare a variable to several integral values

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax:

```
switch (n)
{
   case 1: // code to be executed if n = 1;
      break;
   case 2: // code to be executed if n = 2;
      break;
   default: // code to be executed if n doesn't match any cases
}
```

Important Points about Switch Case Statements:

1. The expression provided in the switch should result in a **constant value** otherwise it would not be valid.

Valid expressions for switch:

```
// Constant expressions allowed
switch(1+2+23)
switch(1*2+3%4)
// Variable expression are allowed provided
// they are assigned with fixed values
```

switch(a*b+c*d)

switch(a+b+c)

2. Duplicate case values are not allowed.

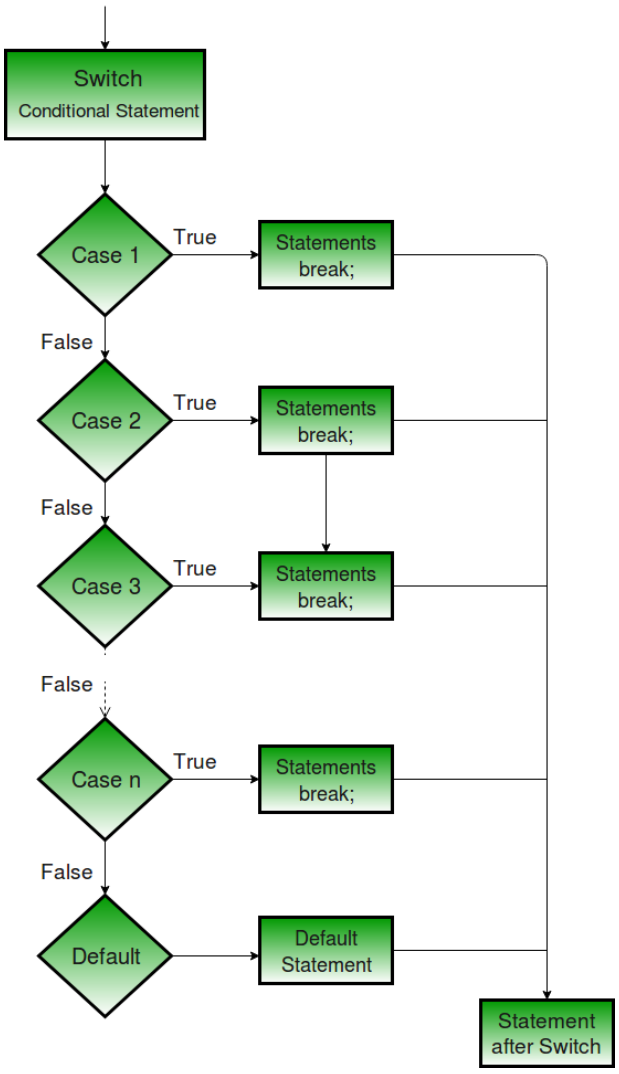
3. The default statement is optional. Even if the switch case statement do not have a default statement, it would run without any problem.

4. The break statement is used inside the switch to terminate a statement sequence. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

5. The break statement is optional. If omitted, execution will continue on into the next case. The flow of control will fall through to subsequent cases until a break is reached.

6. Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

Flowchart:



Example:

```
include <iostream>
using namespace std;

int main() {
int x = 2;
   switch (x)
   {
      case 1:
         cout << "Choice is 1";
         break;
      case 2:
```

```
            cout << "Choice is 2";
            break;
        case 3:
            cout << "Choice is 3";
            break;
        default:
            cout << "Choice other than 1, 2 and 3";
            break;
    }
return 0;
}
```

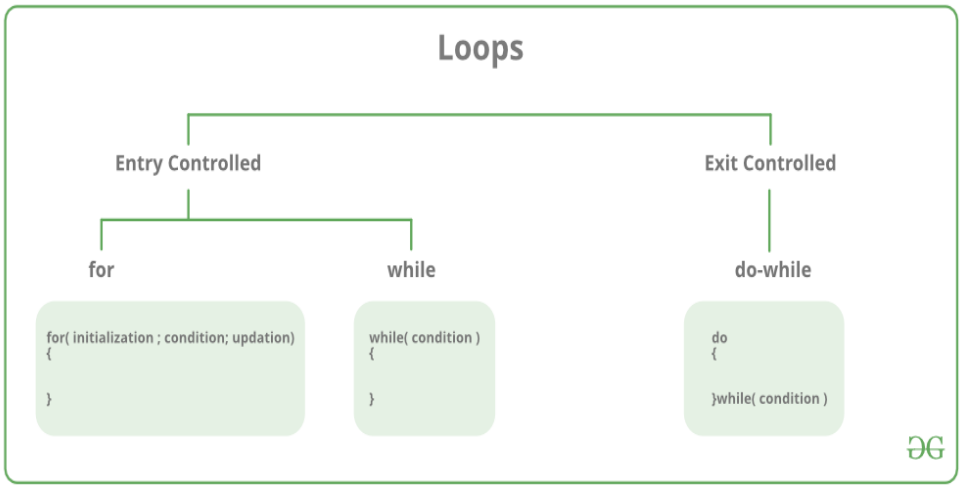
Output:
Choice is 2

❖ Loops in C++

Loops in programming come into use when we need to repeatedly execute a block of statements.

There are mainly two types of loops:

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



❖ For, while, do statement

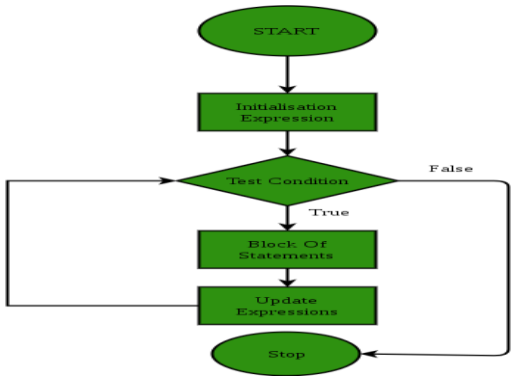
A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

Syntax:

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

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

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



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

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

   return 0;
}
```

Output:

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

While Loop

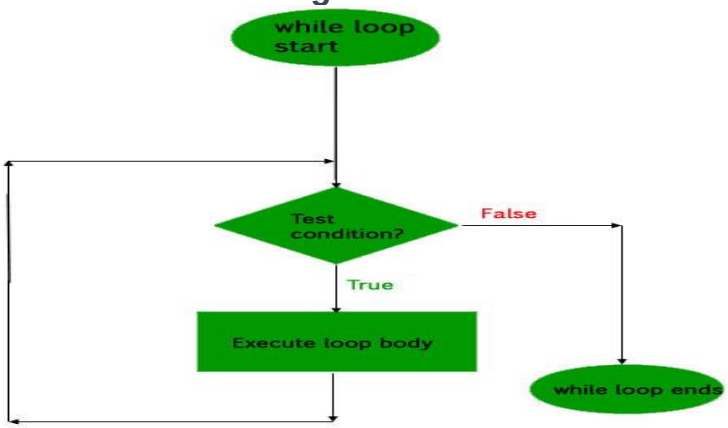
While studying **for loop** we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of test condition.

Syntax:

We have already stated that a loop is mainly consisted of three statements – initialization expression, test expression, update expression. The syntax of the three loops – For, while and do while mainly differs on the placement of these three statements.

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

  update_expression;
}
```

Flow Diagram:



Example:

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

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

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

      // update expression
      i++;
   }

   return 0;
}
```

Output:

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

do while loop

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

controlled loops. **Note**: In do while loop the loop body will execute at least once irrespective of test condition.

Syntax:

initialization expression;

do

{

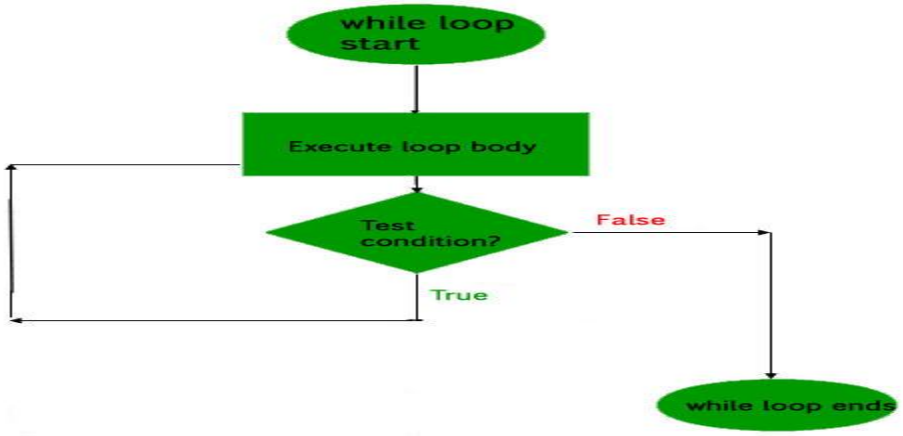
// statements

update_expression;

} while (**test_expression**);

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

Flow Diagram:



Example:

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

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

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

      // update expression
      i++;
```

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

    return 0;
}
```

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

❖ Functions in C++

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

The general form of a function is:

```
return_type function_name([ arg1_type arg1_name, ... ])
{ code }
```

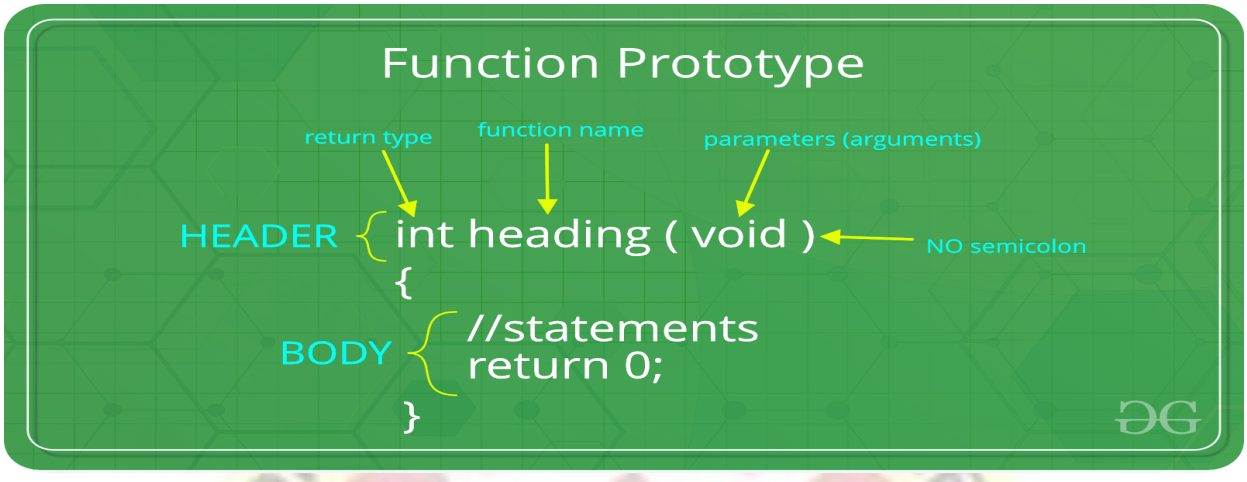
Why do we need functions?

- 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 change at one place if we make future changes to the functionality.
- Functions make code modular. Consider a big file having many lines of codes. 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 working.

Function Declaration

A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the

definition. Below are an example of function declarations. (parameter names are not there in below declarations)




```
// A function that takes two integers as parameters
// and returns an integer
int max(int, int);

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

// A function that takes a char as parameters
// and returns an reference variable
char *call(char b);

// A function that takes a char and an int as parameters
// and returns an integer
int fun(char, int);
```

Parameter Passing to functions: The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters. The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters. There are two most popular ways to pass parameters.

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

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

For example. in the below code, value of x is not modified using the function fun().

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

void fun(int x) {
   x = 30;
}

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

Output:
x = 20

Main Function: The main function is a special function. Every C++ program must contain a function named main. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

Types of main Function:

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

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

2) Second type is main function with parameters :

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

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

When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named argv. Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

❖ Inline functions

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small. The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
// function code
}
```

Inline functions provide following advantages:

1) Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
} //Output: The cube of 3 is: 27
```

❖ Function overloading

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading "Function" name should be the same and the arguments should be different.Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
```

```
  print("ten");
  return 0;
}
```

Output:
Here is int 10
Here is float 10.1
Here is char* ten

How Function Overloading works?

- *Exact match*:- (Function name and Parameter)
- *If* a *not exact match is found:–*

  ->Char, Unsigned char, and short are promoted to an int.

  ->Float is promoted to double

- *If no match found*:

  ->C++ tries to find a match through the standard conversion.

- *ELSE ERROR*

UNIT II

❖ Classes and objects

Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

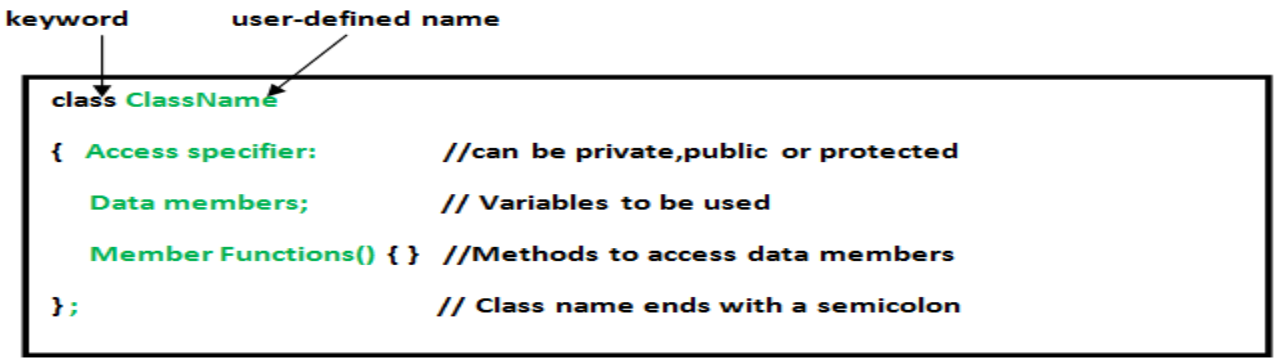
- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in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

Object: 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.

❖ Declaring objects and defining member functions

Defining Class and Declaring Objects

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



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

Syntax:

ClassName ObjectName;

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

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

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

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

   // Data Members
   string geekname;

   // Member Functions()
   void printname()
   {
      cout << "Geekname is: " << geekname;
   }
};

int main() {

   // Declare an object of class geeks
   Geeks obj1;

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

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

Output:
Geekname is: Abhi

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

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

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

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

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

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

// Definition of printname using scope resolution operator ::
void Geeks::printname()
{
   cout << "Geekname is: " << geekname;
}
int main() {

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

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

   // call printid()
```

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

Output:
Geekname is: xyz
Geek id is: 15

❖ Static Member variables and functions

- **Static variables in a Function**: When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing co-routines in C++ or any other application where previous state of function needs to be stored.

```
// C++ program to demonstrate
// the use of static Static
// variables in a Function
#include <iostream>
#include <string>
using namespace std;

void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and
    // will be carried to next
    // function calls
    count++;
}

int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

Output:
0 1 2 3 4

In the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

- **Static variables in a class**: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects.** There can not be multiple copies of same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

```
// C++ program to demonstrate static
// variables inside a class

#include<iostream>
using namespace std;

class GfG
{
  public:
    static int i;

    GfG()
    {
      // Do nothing
    };
};

int main()
{
 GfG obj1;
 GfG obj2;
 obj1.i =2;
 obj2.i = 3;

 // prints value of i
 cout << obj1.i<<" "<<obj2.i;
}
```

We can see in the above program that we have tried to create multiple copies of the static variable i for multiple objects. But this didn't happen. So, a static variable inside a class

should be initialized explicitly by the user using the class name and scope resolution operator outside the class as shown below:

```
// C++ program to demonstrate static
// variables inside a class

#include<iostream>
using namespace std;

class GfG
{
public:
   static int i;

   GfG()
   {
      // Do nothing
   };
};

int GfG::i = 1;

int main()
{
   GfG obj;
   // prints value of i
   cout << obj.i;
}
```

Output:1

Static Members of Class

- **Class objects as static:**

  Just like variables, objects also when declared as static have a scope till the lifetime of program. Consider the below program where the object is non-static.

```
// CPP program to illustrate
// when not using static keyword
#include<iostream>
using namespace std;

class GfG
{
   int i;
   public:
```

```
    GfG()
    {
       i = 0;
       cout << "Inside Constructor\n";
    }
    ~GfG()
    {
       cout << "Inside Destructor\n";
    }
};

int main()
{
   int x = 0;
   if (x==0)
   {
      GfG obj;
   }
   cout << "End of main\n";
}
```

Output:
Inside Constructor
Inside Destructor
End of main

- In the above program the object is declared inside the if block as non-static. So, the scope of variable is inside the if block only. So when the object is created the constructor is invoked and soon as the control of if block gets over the destructor is invoked as the scope of object is inside the if block only where it is declared. Let us now see the change in output if we declare the object as static.

```
// CPP program to illustrate
// class objects as static
#include<iostream>
using namespace std;

class GfG
{
   int i = 0;

   public:
   GfG()
```

```
    {
        i = 0;
        cout << "Inside Constructor\n";
    }

    ~GfG()
    {
        cout << "Inside Destructor\n";
    }
};

int main()
{
    int x = 0;
    if (x==0)
    {
        static GfG obj;
    }
    cout << "End of main\n";
}
```

Output:
Inside Constructor
End of main
Inside Destructor

We can clearly see the change in output. Now the destructor is invoked after the end of main. This happened because the scope of static object is through out the life time of program.

Static functions in a class: Just like the static data members or static variables inside the class, static member functions also does not depend on object of class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator. **Static member functions are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.

```
// C++ program to demonstrate static
// member function in a class
#include<iostream>
using namespace std;
```

```
class GfG
{
  public:

    // static member function
    static void printMsg()
    {
       cout<<"Welcome to GfG!";
    }
};

// main function
int main()
{
    // invoking a static member function
    GfG::printMsg();
}
```

Output:
Welcome to GfG!

❖ Array of objects

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

The syntax for declaring an array of objects is

```
class_name array_name [size] ;
```

To understand the concept of an array of objects, consider this example.

Example : A program to demonstrate the concept of array of objects

```
#include<iostream>
using namespace std;
class books {
  char tit1e [30];
  float price ;
  public:
     void getdata ();
```

```
    void putdata ();
} ;
void books :: getdata () {
  cout<<"Title:”;
  cin>>title;
  cout<<"Price:”;
  cin>>price;
}
void books :: putdata () {
  cout<<"Title:"<<title<< "\n";
  cout<<"Price:"<<price<< "\n”;
  const int size=3 ;
}
int main() {
  books book[size] ;
  for(int i=0;i<size;i++) {
    cout<<"Enter details o£ book "<<(i+1)<<"\n";
    book[i].getdata();
}
  for(int i=0;i<size;i++) {
    cout<<"\nBook "<<(i+l)<<"\n";
    book[i].putdata() ;
}
   return 0;
}
```

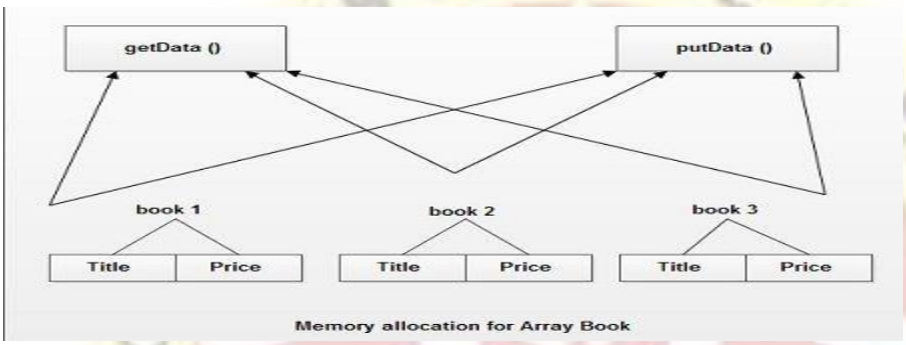
The output of the program is

```
Enter details of book 1
Title: c++
Price: 325
Enter details of book 2
Title: DBMS
Price:. 455
Enter details of book 3
Title: Java
Price: 255
Book 1
Title: c++
Price: 325
Book 2
Title: DBMS
Price: 455
Book 3
Title: Java
Price: 255
```

In this example, an array book of the type class books and size three is declared. This implies that book is an array of three objects of the class books. Note that every object in the array book

can access public members of the class in the same way as any other object, that is, by using the dot operator. For example, the statement book [i] . getdata () invokes the getdata () function for the ith element of array book.

When an array of objects is declared, the memory is allocated in the same way as to multidimensional arrays. For example, for the array book, a separate copy of title and price is created for each member book[0], book[l] and book[2]. However, member functions are stored at a different place in memory and shared among all the array members. For instance, the memory space is allocated to the the array of objects book of the class books



Memory allocation for Array Book

❖ Friend functions

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:

a) A member of another class

b) A global function

```
class Node {
private:
   int key;
   Node* next;

   /* Other members of Node Class */
   friend int LinkedList::search();
   // Only search() of linkedList
   // can access internal members
};
```

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.

3) Friendship is not inherited

A simple and complete C++ program to demonstrate friend Class

```
#include <iostream>
class A {
private:
   int a;

public:
   A() { a = 0; }
   friend class B; // Friend Class
};

class B {
private:
   int b;

public:
   void showA(A& x)
   {
      // Since B is friend of A, it can access
      // private members of A
      std::cout << "A::a=" << x.a;
   }
};

int main()
{
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

Output:

```
A::a=0
```

A simple and complete C++ program to demonstrate friend function of another class

```
#include <iostream>

class B;

class A {
public:
   void showB(B&);
};

class B {
private:
   int b;

public:
   B() { b = 0; }
   friend void A::showB(B& x); // Friend function
};

void A::showB(B& x)
{
   // Since showB() is friend of B, it can
   // access private members of B
   std::cout << "B::b = " << x.b;
}

int main()
{
   A a;
   B x;
   a.showB(x);
   return 0;
}
```

Output:

```
B::b = 0
```

❖ Overloading member functions

we can have multiple definitions for the same **function** name in the same scope. The definition of the **function** must differ from each other by the types and/or the number of arguments in the argument list. You cannot **overload function** declarations that differ only by return type.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

❖ Bit fields and classes

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

Syntax

declarator **:** *constant-expression*

The (optional) *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number

of bits the member occupies in the structure. Anonymous bit fields — that is, bit-field members with no identifier — can be used for padding.

The following example declares a structure that contains bit fields

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
   unsigned short nWeekDay  : 3;    // 0..7   (3 bits)
   unsigned short nMonthDay : 6;    // 0..31  (6 bits)
   unsigned short nMonth    : 5;    // 0..12  (5 bits)
   unsigned short nYear     : 8;    // 0..100 (8 bits)
};
```

The following list details erroneous operations on bit fields:

- Taking the address of a bit field.
- Initializing a non-**const** reference with a bit field.

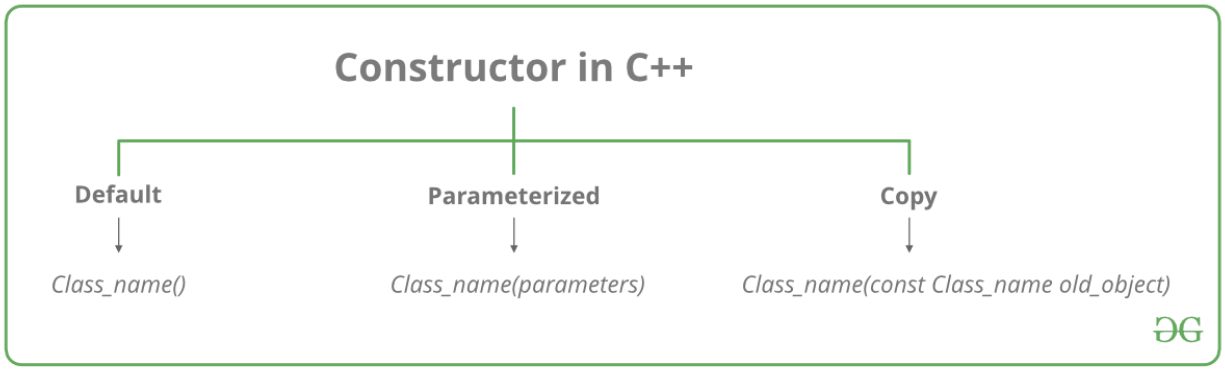
❖ Constructor and Destructor with static members

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object (instance of class) create. It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).



Types of Constructors

1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// concept of Constructors
#include <iostream>
using namespace std;

class construct
{
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
{
   // Default constructor called automatically
   // when the object is created
   construct c;
   cout << "a: " << c.a << endl
        << "b: " << c.b;
   return 1;
}
```

Output:
a: 10
b: 20

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// parameterized constructors
#include <iostream>
using namespace std;

class Point
{
private:
   int x, y;

public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }

   int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};

int main()
{
   // Constructor called
   Point p1(10, 15);
```

```
    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call
```

```
Example e(0, 50);          // Implicit call
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

3. Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.

Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

```
#include "iostream"
using namespace std;

class point
{
private:
  double x, y;
```

```
public:

  // Non-default Constructor &
  // default Constructor
  point (double px, double py)
  {
    x = px, y = py;
  }
};

int main(void)
{

  // Define an array of size
  // 10 & of type point
  // This line will cause error
  point a[10];

  // Remove above line and program
  // will compile without error
  point b = point(5, 6);
}
```

Output:
Error: point (double px, double py): expects 2 arguments, 0 provided

DESTRUCTOR

Destructor is a member function which destructs or deletes an object.

Syntax:

~constructor-name();

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

(1) the function ends

(2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~) .

Destructors don't take any argument and don't return anything.

```
class String {
private:
   char* s;
   int size;

public:
   String(char*); // constructor
   ~String(); // destructor
};

String::String(char* c)
{
   size = strlen(c);
   s = new char[size + 1];
   strcpy(s, c);
}
String::~String() { delete[] s; }
```

Static members in C++

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

class A
{
public:
   A() { cout << "A's Constructor Called " << endl;  }
};

class B
{
```

```
   static A a;
public:
   B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
   B b;
   return 0;
}
```

Output:
B's Constructor Called

The above program calls only B's constructor, it doesn't call A's constructor. The reason for this is simple, *static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator*.
If we try to access static member 'a' without explicit definition of it, we will get compilation error. For example, following program fails in compilation.

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

class A
{
   int x;
public:
   A() { cout << "A's constructor called " << endl;  }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

int main()
{
   B b;
   A a = b.getA();
   return 0;
}
```

Output:
Compiler Error: undefined reference to `B::a'

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

class A
{
   int x;
public:
   A() { cout << "A's constructor called " << endl;  }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

A B::a;  // definition of a

int main()
{
   B b1, b2, b3;
   A a = b1.getA();

   return 0;
}
```

Output:
A's constructor called
B's constructor called
B's constructor called
B's constructor called

Note that the above program calls B's constructor 3 times for 3 objects (b1, b2 and b3), but calls A's constructor only once. The reason is, *static members are shared among all objects. That is why they are also known as class members or class fields*. Also, *static members can be accessed without any object*, see the below program where static member 'a' is accessed without any object.

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

```
class A
{
   int x;
public:
   A() { cout << "A's constructor called " << endl;  }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

A B::a;  // definition of a

int main()
{
   // static member 'a' is accessed without any object of B
   A a = B::getA();

   return 0;
}
```

Output:
A's constructor called

UNIT III

❖ Operator Overloading

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different task.

This can be done by declaring the function, its syntax is,

```
Return_Type classname :: operator op(Argument list)
{
   Function Body
```

}

In the above syntax Return_Type is value type to be returned to another object, operator op is the function where the operator is a keyword and op is the operator to be overloaded.

Operator function must be either non-static (member function) or friend function.

Operator Overloading can be done by using **three approaches**, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Below are some criteria/rules to define the operator function:

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
- All the class member object should be public if operator overloading is implemented.
- Operators that cannot be overloaded are **. .* :: ?:**
- Operator cannot be used to overload when declaring that function as friend function = **() []** ->.

❖ Overloading unary operator

Overloading Unary Operator: Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is a overloading of an operator operating on a single operand.

Example:

Assume that class Distance takes two member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type).

```
// C++ program to show unary operator overloading
#include <iostream>

using namespace std;
```

```
class Distance {
public:

   // Member Object
   int feet, inch;

   // Constructor to initialize the object's value
   Distance(int f, int i)
   {
      this->feet = f;
      this->inch = i;
   }

   // Overloading(-) operator to perform decrement
   // operation of Distance object
   void operator-()
   {
      feet--;
      inch--;
      cout << "\nFeet & Inches(Decrement): " << feet << "'" << inch;
   }
};

// Driver Code
int main()
{
   // Declare and Initialize the constructor
   Distance d1(8, 9);

   // Use (-) unary operator by single operand
   -d1;
   return 0;
}
```

Output:
Feet & Inches(Decrement): 7'8

In the above program, it shows that no argument is passed and no return_type value is returned, because unary operator works on a single operand. (-) operator change the functionality to its member function.

❖ Overloading Binary operators

In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

Let's take the same example of class Distance, but this time, add two distance objects.

```
// C++ program to show binary operator overloading
#include <iostream>

using namespace std;

class Distance {
public:
   // Member Object
   int feet, inch;
   // No Parameter Constructor
   Distance()
   {
      this->feet = 0;
      this->inch = 0;
   }

   // Constructor to initialize the object's value
   // Parametrized Constructor
   Distance(int f, int i)
   {
      this->feet = f;
      this->inch = i;
   }

   // Overloading (+) operator to perform addition of
   // two distance object
   Distance operator+(Distance& d2) // Call by reference
   {
      // Create an object to return
      Distance d3;

      // Perform addition of feet and inches
      d3.feet = this->feet + d2.feet;
      d3.inch = this->inch + d2.inch;

      // Return the resulting object
      return d3;
   }
```

```
};

// Driver Code
int main()
{
   // Declaring and Initializing first object
   Distance d1(8, 9);

   // Declaring and Initializing second object
   Distance d2(10, 2);

   // Declaring third object
   Distance d3;

   // Use overloaded operator
   d3 = d1 + d2;

   // Display the result
   cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
   return 0;
}
```

Output:

Total Feet & Inches: 18'11

Here in the above program, Distance operator+(Distance &d2), here return type of function is distance and it uses call by references to pass an argument. d3 = d1 + d2; here, d1 calls the operator function of its class object and takes d2 as a parameter, by which operator function return object and the result will reflect in the d3 object.

❖ Overloading Friend functions

In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.

Let’s take the same example using the friend function.

```
// C++ program to show binary operator overloading
#include <iostream>

using namespace std;

class Distance {
public:

   // Member Object
   int feet, inch;

   // No Parameter Constructor
   Distance()
   {
      this->feet = 0;
      this->inch = 0;
   }

   // Constructor to initialize the object's value
   // Parametrized Constructor
   Distance(int f, int i)
   {
      this->feet = f;
      this->inch = i;
   }

   // Declaring friend function using friend keyword
   friend Distance operator+(Distance&, Distance&);
};

// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
   // Create an object to return
   Distance d3;

   // Perform addition of feet and inches
   d3.feet = d1.feet + d2.feet;
   d3.inch = d1.inch + d2.inch;

   // Return the resulting object
   return d3;
}

// Driver Code
int main()
```

```
{
   // Declaring and Initializing first object
   Distance d1(8, 9);

   // Declaring and Initializing second object
   Distance d2(10, 2);

   // Declaring third object
   Distance d3;

   // Use overloaded operator
   d3 = d1 + d2;

   // Display the result
   cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
   return 0;
}
```

Output:

Total Feet & Inches: 18'11

Here in the above program, operator function is implemented outside of class scope by declaring that function as the friend function.

❖ Type conversion

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.
   - bool -> char -> short int -> int ->
   - unsigned int -> long -> unsigned ->

     long long -> float -> double -> long double

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

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

int main()
{
   int x = 10; // integer x
   char y = 'a'; // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.0;

   cout << "x = " << x << endl
        << "y = " << y << endl
        << "z = " << z << endl;

   return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

   In C++, it can be done by two ways:

   - **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

     Syntax:

     (type) expression

     where *type* indicates the data type to which the final result is converted.

     Example:

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
using namespace std;

int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   cout << "Sum = " << sum;

   return 0;
}
```

Output:
Sum = 2

- **Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type. C++ supports four types of casting:

1. Static Cast
2. Dynamic Cast
3. Const Cast
4. Reinterpret Cast

Example:

```
#include <iostream>
using namespace std;
int main()
{
   float f = 3.5;

   // using cast operator
   int b = static_cast<int>(f);

   cout << b;
}
```

Output: 3

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

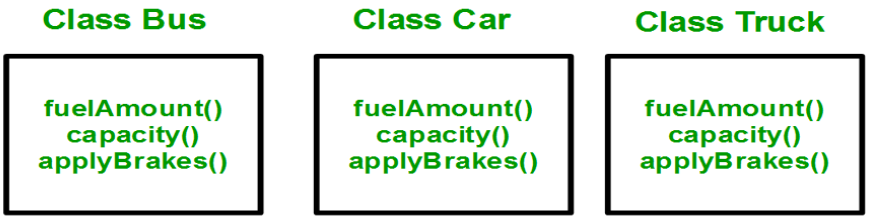
❖ Inheritance and its types introduction

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

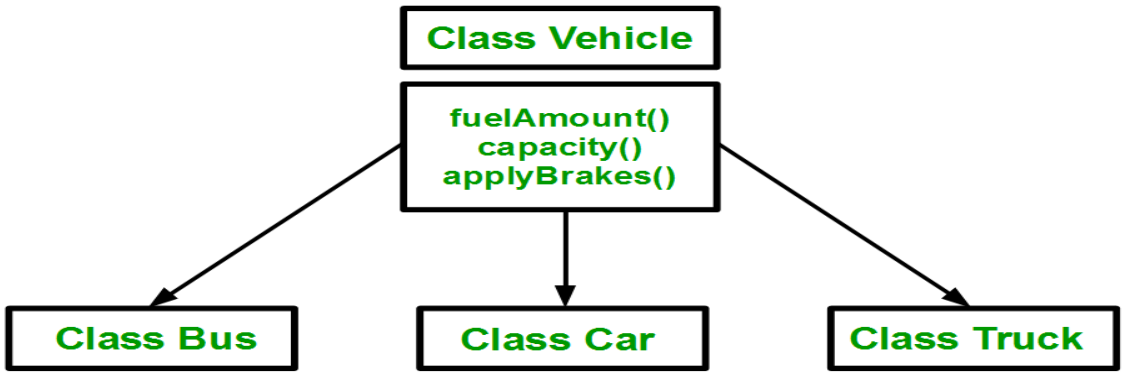
Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.

Syntax:

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

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub 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.

```
// C++ program to demonstrate implementation
// of Inheritance

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

//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
```

```
class Child : public Parent
{
   public:
     int id_c;
};

//main function
int main()
  {

      Child obj1;

      // An object of class child has all data members
      // and member functions of class parent
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj1.id_c << endl;
      cout << "Parent id is " <<  obj1.id_p << endl;

      return 0;
  }
```

Output:

```
Child id is 7
Parent id is 91
```

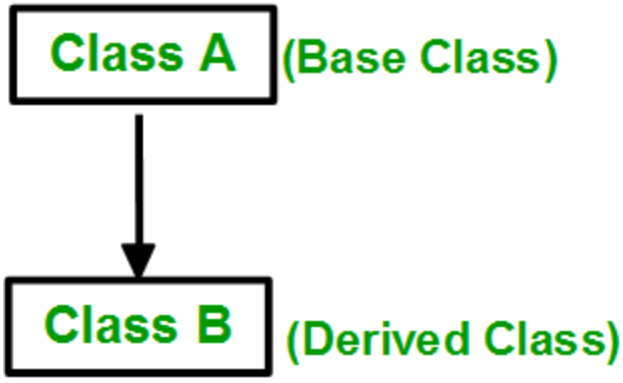
In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

Modes of Inheritance

1. **Public mode**: If we derive a sub class 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 derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

❖ Single inheritance

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

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

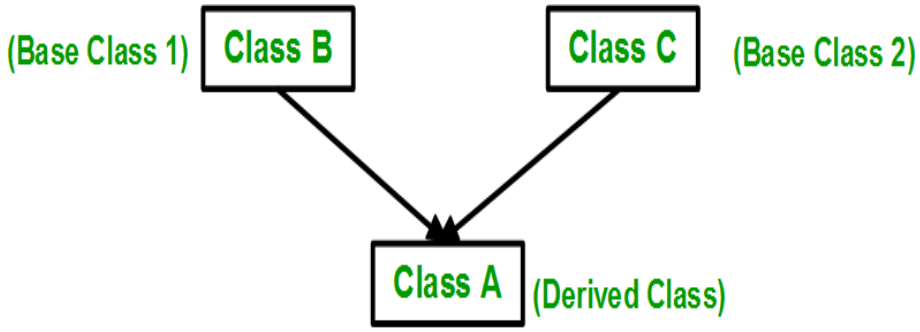
```
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from a single base classes
class Car: public Vehicle{

};

// 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

❖ Multiple Inheritance

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



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 access mode for every base class must be specified.

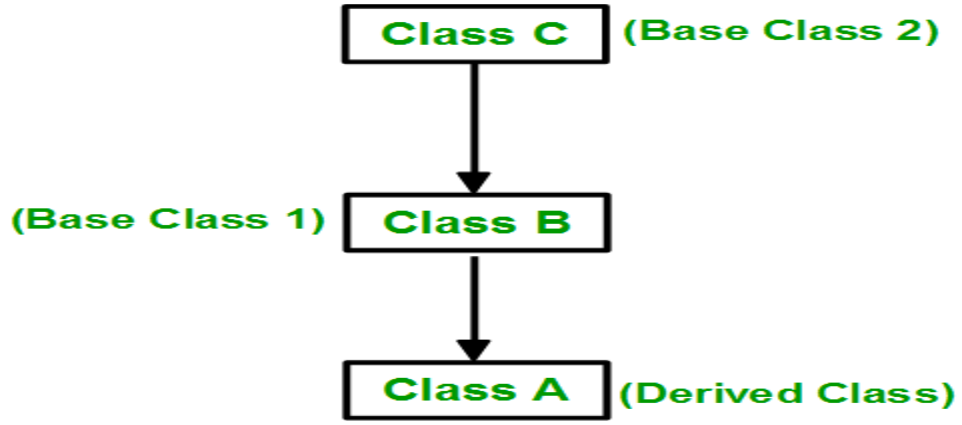
```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
```

```
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// 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 Vehicle

❖ Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class.




```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
```

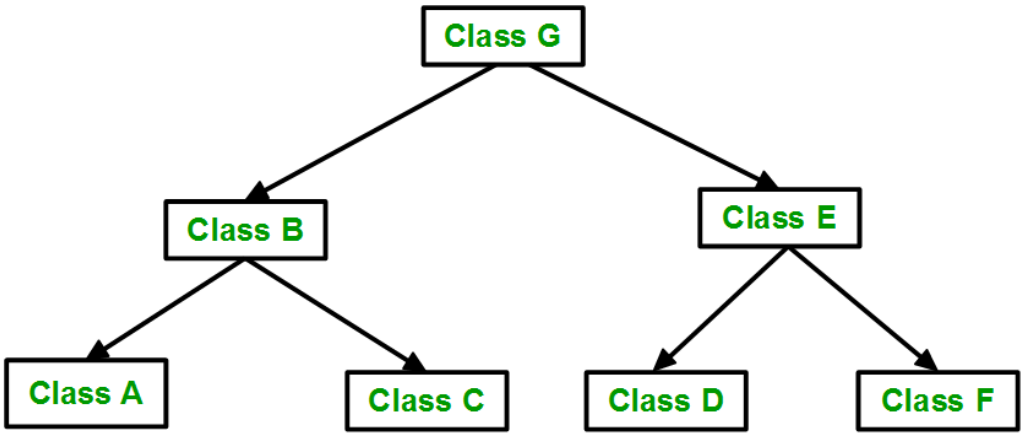
```
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from the derived base class fourWheeler
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

// 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
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

❖ Hierarchical Inheritance

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



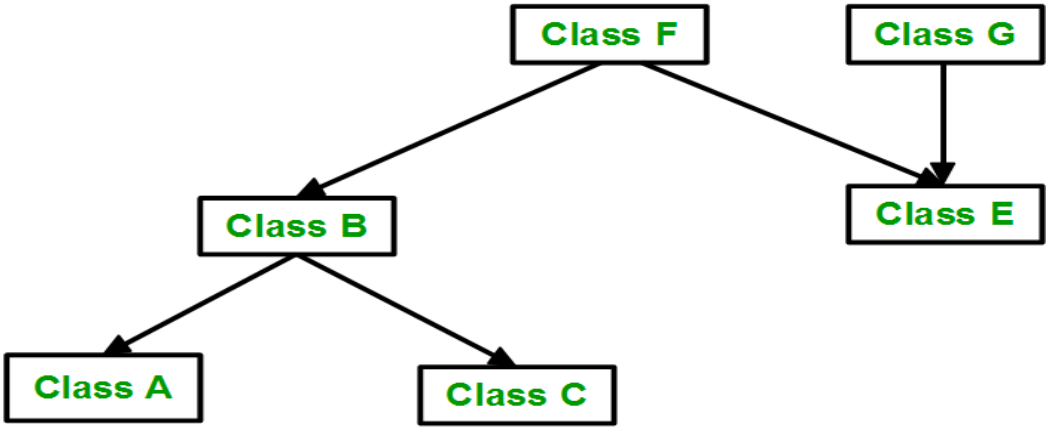

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// 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 is a Vehicle

❖ Hybrid inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:




```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

//base class
class Fare
```

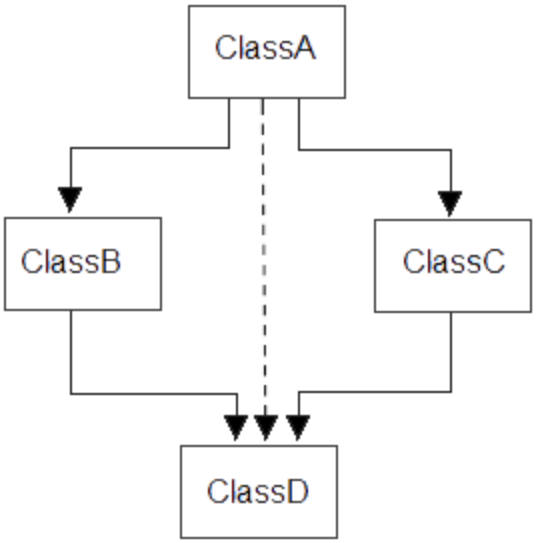
```
{
   public:
   Fare()
   {
      cout<<"Fare of Vehicle\n";
   }
};

// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle, public Fare
{

};

// 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
```

❖ Multipath inheritance

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



Consider the following program:

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <conio.h>
#include <iostream.h>
class ClassA {
public:
   int a;
};

class ClassB : public ClassA {
public:
   int b;
};
class ClassC : public ClassA {
public:
   int c;
};

class ClassD : public ClassB, public ClassC {
public:
   int d;
};

void main()
{

   ClassD obj;
```

```
    // obj.a = 10;              //Statement 1, Error
    // obj.a = 100;             //Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n A from ClassB  : " << obj.ClassB::a;
    cout << "\n A from ClassC  : " << obj.ClassC::a;

    cout << "\n B : " << obj.b;
    cout << "\n C : " << obj.c;
    cout << "\n D : " << obj.d;
}
```

Output:

```
A from ClassB  : 10
A from ClassC  : 100
B : 20
C : 30
D : 40
```

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

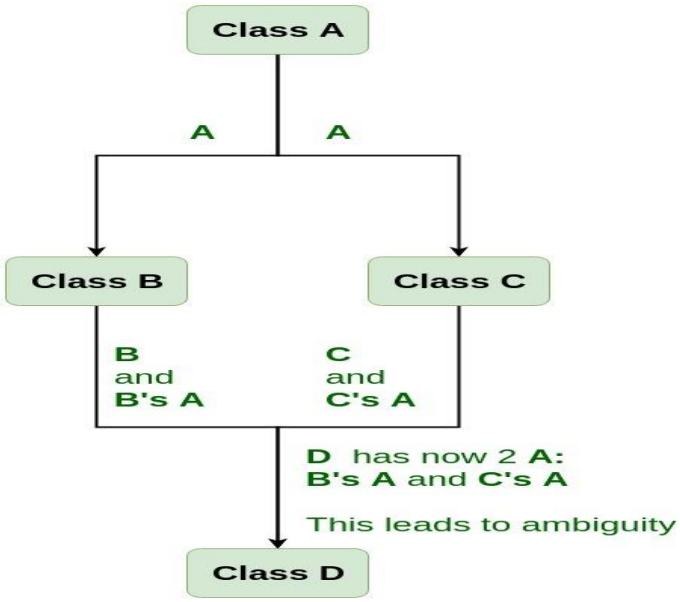
❖ Virtual Base Classes

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:

Consider the situation where we have one class **A** .This class is **A** is inherited by two other

classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Syntax for Virtual Base Classes:

Syntax 1:

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

Syntax 2:

```
class C : public virtual A
{
};
```

Note: virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members

is shared by all the base classes that use virtual base.

Example 1

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

class A {
public:
   int a;
   A() // constructor
   {
      a = 10;
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
   D object; // object creation of class d
   cout << "a = " << object.a << endl;

   return 0;
}
```

Output:

a = 10

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

❖ Abstract Classes.

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have

implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

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

  /* Other members */
};
```

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

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

class Base
{
  int x;
public:
   virtual void fun() = 0;
   int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
   int y;
public:
   void fun() { cout << "fun() called"; }
};

int main(void)
{
   Derived d;
   d.fun();
   return 0;
}
```

Output:
fun() called

UNIT IV

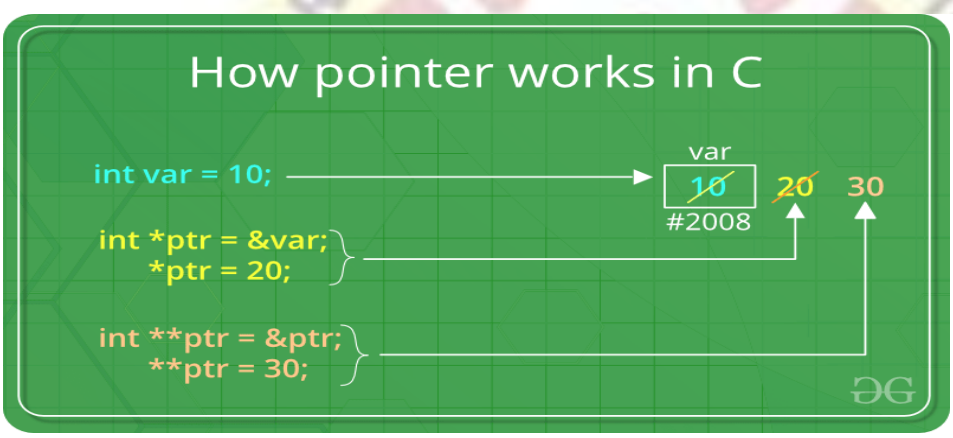
❖ Pointers and Declaration

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C++ has the format:

Syntax:

```
datatype *var_name;
```

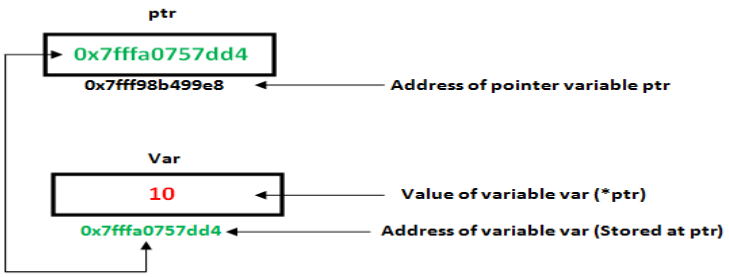
```
int *ptr;   //ptr can point to an address which holds int data
```



How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.




```
// C++ program to illustrate Pointers in C++

#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;

        //declare pointer variable
    int *ptr;

    //note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
//Driver program
int main()
{
    geeks();
}
```

Output:

```
Value at ptr = 0x7ffcb9e9ea4c
Value at var = 20
Value at *ptr = 20
```

❖ Pointer to class, object this pointer

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator -> operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

```
#include <iostream>

using namespace std;

class Box {
  public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
  Box *ptrBox;                // Declare pointer to a class.

  // Save the address of first object
  ptrBox = &Box1;

  // Now try to access a member using member access operator
  cout << "Volume of Box1: " << ptrBox->Volume() << endl;

  // Save the address of second object
  ptrBox = &Box2;

  // Now try to access a member using member access operator
  cout << "Volume of Box2: " << ptrBox->Volume() << endl;

  return 0;
}
```

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

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

This Pointer

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Each object gets its own copy of data members and all objects share a single copy of member functions. The compiler supplies an implicit pointer along with the names of the functions as 'this'. The 'this' pointer is passed as a hidden argument to all non static member function calls and is available as a local variable within the body of all non static functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name). For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X * .

C++ lets object destroy themselves by calling the following code :

```
delete this;
```

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

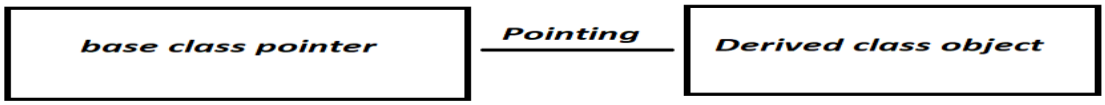
2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
  // Some processing
  return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

❖ Pointers to derived classes and base classes

The pointer of Base Class pointing different object of derived class:



Used to point the derived class object and pointer can still use aspects of bsse class

Approach:

- A derived class is a class which takes some properties from its base class.
- It is true that a pointer of one class can point to other class, but classes must be a base and derived class, then it is possible.
- To access the **variable of the base class**, base class pointer will be used.
- So, a pointer is type of base class, and it can access all, public function and variables of base class since pointer is of base class, this is known as binding pointer.
- In this pointer base class is owned by base class but points to derived class object.
- Same works with derived class pointer, values is changed.

Below is the C++ program to illustrate the implementation of the base class pointer pointing to the derived class.

```
// C++ program to illustrate the
// implementation of the base class
// pointer pointing to derived class
#include <iostream>
using namespace std;

// Base Class
class BaseClass {
public:
    int var_base;

    // Function to display the base
    // class members
    void display()
    {
        cout << "Displaying Base class"
             << " variable var_base: " << var_base << endl;
```

```
    }
};

// Class derived from the Base Class
class DerivedClass : public BaseClass {
public:
    int var_derived;

    // Function to display the base
    // and derived class members
    void display()
    {
        cout << "Displaying Base class"
             << "variable var_base: " << var_base << endl;
        cout << "Displaying Derived "
             << " class variable var_derived: "
             << var_derived << endl;
    }
};

// Driver Code
int main()
{
    // Pointer to base class
    BaseClass* base_class_pointer;
    BaseClass obj_base;
    DerivedClass obj_derived;

    // Pointing to derived class
    base_class_pointer = &obj_derived;

    base_class_pointer->var_base = 34;

    // Throw an error
    base_class_pointer->display();

    base_class_pointer->var_base = 3400;
    base_class_pointer->display();

    DerivedClass* derived_class_pointer;
    derived_class_pointer = &obj_derived;
    derived_class_pointer->var_base = 9448;
    derived_class_pointer->var_derived = 98;
    derived_class_pointer->display();

    return 0;
```

}

Output:
Displaying Base class variable var_base: 34
Displaying Base class variable var_base: 3400
Displaying Base classvariable var_base: 9448
Displaying Derived class variable var_derived: 98

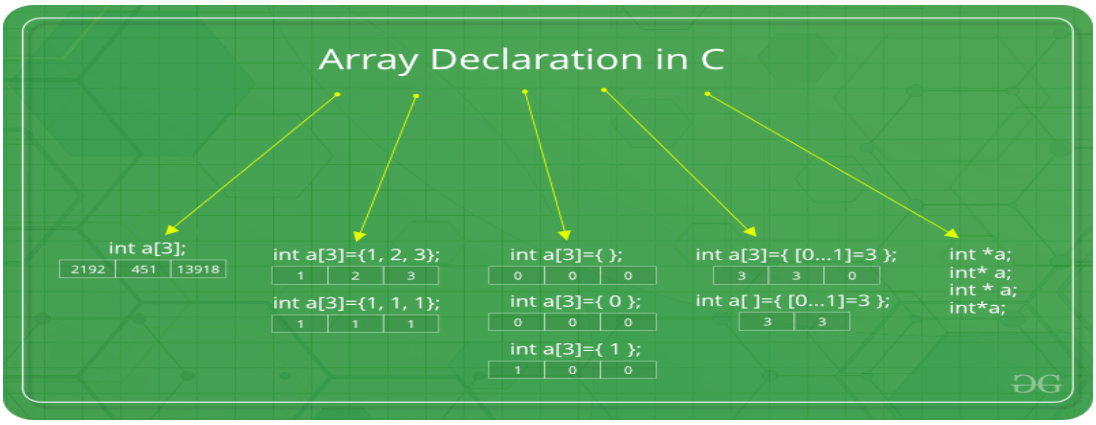
Conclusion:

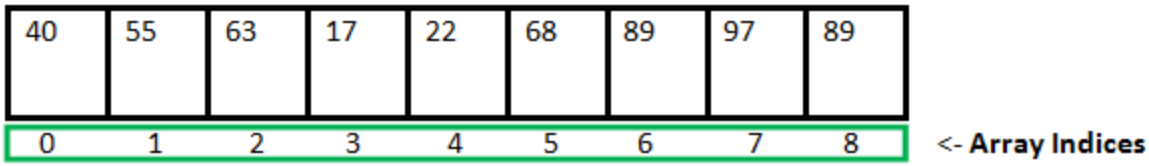
- A pointer to derived class is a pointer of base class pointing to derived class, but it will hold its aspect.
- This pointer of base class will be able to temper functions and variables of its own class and can still point to derived class object.

❖ Arrays and characteristics

An array in C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C++ can store derived data types such as the structures, pointers etc. Given below is the picture representation of an array.

Array declaration in C++:





Array Length = 9
First Index = 0
Last Index = 8

Advantages of an Array in C/C++:

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Facts about Array in C/C++:

- **Accessing Array Elements:**

  Array elements are accessed by using an integer index. Array index starts with 0 and goes

till size of array minus 1.

- Name of the array is also a pointer to the first element of array.

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

int main()
{
    int arr[5];
    arr[0] = 5;
    arr[2] = -10;

    // this is same as arr[1] = 2
    arr[3 / 2] = 2;
    arr[3] = arr[0];

    cout << arr[0] << " " << arr[1] << " " << arr[2] << " "
         << arr[3];

    return 0;
}
```

Output

-449684907 4195777

❖ Array of classes

- Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don't need to pass size of Array as a separate parameter.
- With C-style array there is more risk of array being decayed into a pointer. Array classes don't decay into pointers
- Array classes are generally more efficient, light-weight and reliable than C-style arrays.

Operations on array :-

1. at() :- This function is used to access the elements of array.

2. get() :- This function is also used to access the elements of array. This function is not the member of array class but overloaded function from class tuple.

3. operator[] :- This is similar to C-style arrays. This method is also used to access array elements.

```
// C++ code to demonstrate working of array,
// to() and get()
#include<iostream>
#include<array> // for array, at()
#include<tuple> // for get()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing array elements using at()
   cout << "The array elements are (using at()) : ";
   for ( int i=0; i<6; i++)
   cout << ar.at(i) << " ";
   cout << endl;

   // Printing array elements using get()
   cout << "The array elements are (using get()) : ";
   cout << get<0>(ar) << " " << get<1>(ar) << " ";
   cout << get<2>(ar) << " " << get<3>(ar) << " ";
   cout << get<4>(ar) << " " << get<5>(ar) << " ";
   cout << endl;

   // Printing array elements using operator[]
   cout << "The array elements are (using operator[]) : ";
   for ( int i=0; i<6; i++)
   cout << ar[i] << " ";
   cout << endl;

   return 0;

}
```

Output:
The array elemets are (using at()) : 1 2 3 4 5 6
The array elemets are (using get()) : 1 2 3 4 5 6
The array elements are (using operator[]) : 1 2 3 4 5 6

4. front() :- This returns the first element of array.

5. back() :- This returns the last element of array.

```
// C++ code to demonstrate working of
// front() and back()
```

```
#include<iostream>
#include<array> // for front() and back()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing first element of array
   cout << "First element of array is : ";
   cout << ar.front() << endl;

   // Printing last element of array
   cout << "Last element of array is : ";
   cout << ar.back() << endl;

   return 0;

}
```

Output:
First element of array is : 1
Last element of array is : 6

6. size() :- It returns the number of elements in array. This is a property that C-style arrays lack.

7. max_size() :- It returns the maximum number of elements array can hold i.e, the size with which array is declared. The size() and max_size() return the same value.

```
// C++ code to demonstrate working of
// size() and max_size()
#include<iostream>
#include<array> // for size() and max_size()
using namespace std;
int main()
{
   // Initializing the array elements
   array<int,6> ar = {1, 2, 3, 4, 5, 6};

   // Printing number of array elements
   cout << "The number of array elements is : ";
   cout << ar.size() << endl;

   // Printing maximum elements array can hold
   cout << "Maximum elements array can hold is : ";
```

```
    cout << ar.max_size() << endl;

    return 0;

}
```

Output:

```
The number of array elements is : 6
Maximum elements array can hold is : 6
```

8. swap() :- The swap() swaps all elements of one array with other.

```
// C++ code to demonstrate working of swap()
#include<iostream>
#include<array> // for swap() and array
using namespace std;
int main()
{

    // Initializing 1st array
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Initializing 2nd array
    array<int,6> ar1 = {7, 8, 9, 10, 11, 12};

    // Printing 1st and 2nd array before swapping
    cout << "The first array elements before swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements before swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar1[i] << " ";
    cout << endl;

    // Swapping ar1 values with ar
    ar.swap(ar1);

    // Printing 1st and 2nd array after swapping
    cout << "The first array elements after swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements after swapping are : ";
    for (int i=0; i<6; i++)
    cout << ar1[i] << " ";
    cout << endl;
```

```
    return 0;

}
```

Output:
The first array elements before swapping are : 1 2 3 4 5 6
The second array elements before swapping are : 7 8 9 10 11 12
The first array elements after swapping are : 7 8 9 10 11 12
The second array elements after swapping are : 1 2 3 4 5 6

9. empty() :- This function returns true when the array size is zero else returns false.

10. fill() :- This function is used to fill the entire array with a particular value.

```
// C++ code to demonstrate working of empty()
// and fill()
#include<iostream>
#include<array> // for fill() and empty()
using namespace std;
int main()
{

    // Declaring 1st array
    array<int,6> ar;

    // Declaring 2nd array
    array<int,0> ar1;

    // Checking size of array if it is empty
    ar1.empty()? cout << "Array empty":
       cout << "Array not empty";
    cout << endl;

    // Filling array with 0
    ar.fill(0);

    // Displaying array after filling
    cout << "Array after filling operation is : ";
    for ( int i=0; i<6; i++)
       cout << ar[i] << " ";

    return 0;

}
```

Output:
Array empty
Array after filling operation is : 0 0 0 0 0 0

❖ Memory models- New and delete operators

Memory Mode-Describes the interactions of threads through memory and their shared use of data.

- Tells us if our program has well defined behavior.
- Constrains code generation for compiler

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**.

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is memory allocated/deallocated in C++?

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
- pointer-variable = **new** data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.
Example:
// Pointer initialized with NULL

```
// Then request memory for the variable
int *p = NULL;
p = new int;
        OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

Initialize memory: We can also initialize the memory using new operator:

pointer-variable = **new** data-type(value);

Example:

```
int *p = new int(25);
float *q = new float(75.25);
```

- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.

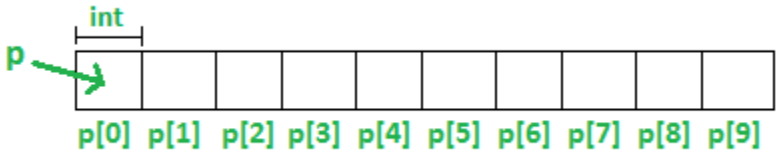
  pointer-variable = **new** data-type[size];

  where size(a variable) specifies the number of elements in an array.

  Example:

  ```
  int *p = new int[10]
  ```

  Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



delete operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
delete p;
delete q;
delete[] pointer-variable;
```

Example:

```
// It will free the entire array
// pointed by p.
delete[] p;
```

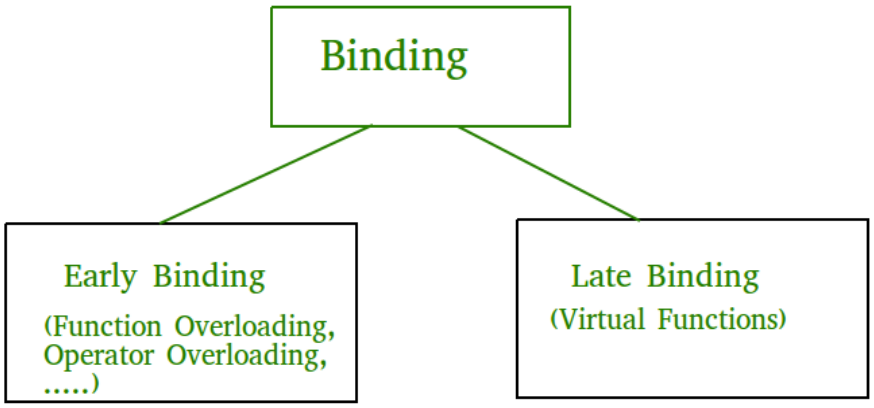
```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new(nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float *r = new float(75.25);

    cout << "Value of r: " << *r << endl;

    // Request block of memory of size n
    int n = 5;
    int *q = new(nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
```

```
        q[i] = i+1;

    cout << "Value store in block of memory: ";
    for (int i = 0; i < n; i++)
        cout << q[i] << " ";
}

// freed the allocated memory
delete p;
delete r;

// freed the block of allocated memory
delete[] q;

return 0;
}
```

Output:
Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5

❖ Dynamic object-Binding

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language

instruction that tells the mainframe to leap to the address of the function.

By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

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

class Base
{
public:
   void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;

   // The function call decided at
   // compile time (compiler sees type
   // of pointer and calls base class
   // function.
   bp->show();

   return 0;
}
```

Output:
In Base

Late Binding : (Run time polymorphism) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition (Refer this for details). This can be achieved by declaring a virtual function.

```
// CPP Program to illustrate late binding
#include<iostream>
using namespace std;
```

```
class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->show();  // RUN-TIME POLYMORPHISM
   return 0;
}
```

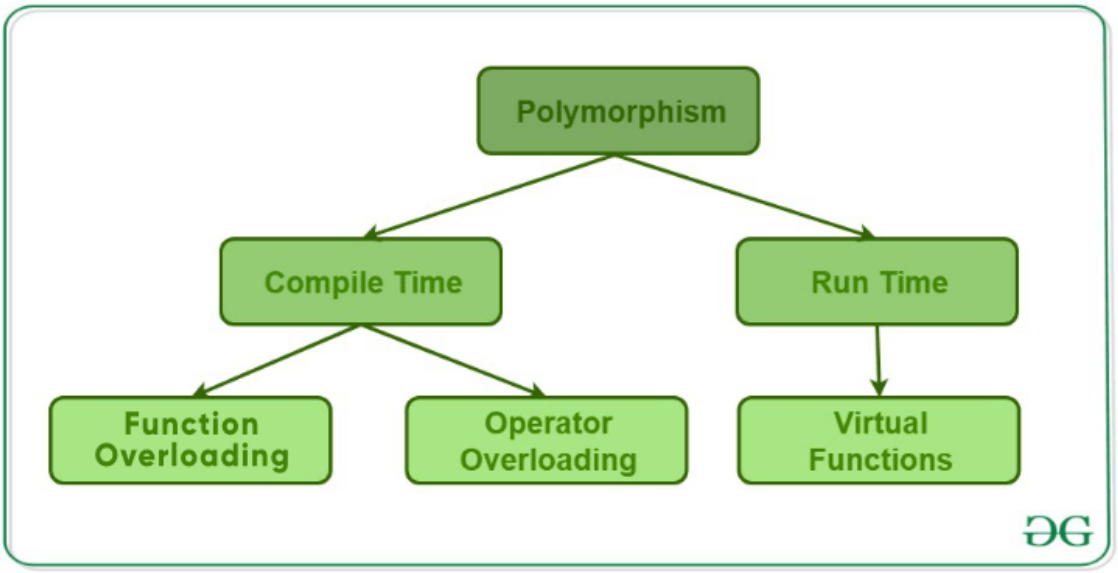
Output:
In Derived

❖ Polymorphism and virtual Functions

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, 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. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

2.Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

- Function overriding on the other hand 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**.

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.

3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
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:
   void print()
   {
      cout << "print derived class" << endl;
   }

   void show()
   {
      cout << "show derived class" << endl;
   }
};

int main()
{
   base* bptr;
```

```
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

print derived class

show base class

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

UNIT V

❖ Files

A **file** can be a data set that you can read and write repeatedly. A **stream** of bytes generated by a program (such as a pipeline). A **stream** of bytes received from or sent to a peripheral device.**Files** are used to store data in a storage device permanently. **File** handling provides a mechanism to store the output of a program in a **file** and to perform various operations on it.

1. **Sequential Access –**

   It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editor and compiler usually access the file in this fashion.

   Read and write make up the bulk of the operation on a file. A read operation -*read next*- read the next position of the file and automatically advance a file pointer, which keeps track

I/O location. Similarly, for the write*write next* append to the end of the file and advance to the newly written material.

Key points:

- Data is accessed one record right after another record in an order.
- When we use read command, it move ahead pointer by one
- When we use write command, it will allocate memory and move the pointer to the end of the file
- Such a method is reasonable for tape.

2. **Direct Access –**

   Another method is *direct access method* also known as *relative access method*. A filed-length logical record that allows the program to read and write record rapidly. in no particular order. The direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59 and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file.

   A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on.

3. **Index sequential method –**

   It is the other method of accessing a file which is built on the top of the sequential access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index and then by the help of pointer we access the file directly.

   Key points:

   - It is built on top of Sequential access.
   - It control the pointer by using index.

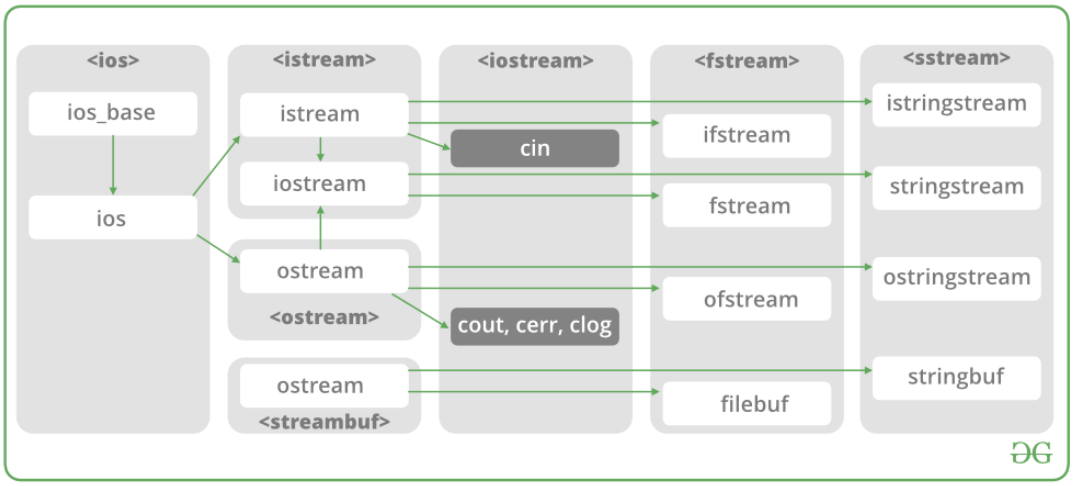
❖ Files stream classes

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.



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");

❖ File modes

Member Constant	Stands For	Access

in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Default Open Modes :

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

Below is the implementation by using **ifsream & 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 (fin) {
```

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

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

    // Close the file
    fin.close();

    return 0;
}
```

❖ Sequential read operations

Sequential access is a term describing a group of elements (such as data in a memory array or a disk **file** or on magnetic tape data storage) being **accessed** in a predetermined, ordered sequence. ... **Sequential access** is sometimes the only way of **accessing** the data, for example if it is on a tape.

SEQUENTIAL INPUT AND OUTPUT OPERATIONS

- The file stream classes support a number of member functions for performing the input and output operations on files.
- Functions like put () and get() are designed for handling a single character at a time whereas write() and read() are designed to write and read blocks of binary data.

Put () and get () functions

- The function put () writes a single character to the associated stream and get () reads a single character from the associated stream.

- Below program shows how these functions work on a file. The program requests for a string and writes it character by character, to the file using the put () function in a for loop. The length of the string is used to terminate the for loop.

- The program then displays the contents of the file on the screen. It uses the function get () to

fetch a character from the file and continues to do so until the end-of-file condition is reached. The character read from the file is displayed on the screen using the operator<<.

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
Void main ()
{
Char string [20];
Cout<<”enter a string”<<endl;
Cin>>string;
}
```

❖ Sequential write operations

Steps to create (or write to) a sequential access file:

Declare a stream variable name: ofstream fout; //each **file** has its own stream buffer. ...

Open the **file**: fout.open("scores.dat", ios::out); fout is the stream variable name previously declared. ...

Write data to the **file**: fout<<grade<<endl; fout<<"Mr. ...

Close the **file**:

❖ Binary Files

Writing

To write a binary file in C++ use write method. It is used to write a given number of bytes on the given stream, starting at the position of the "put" pointer. The file is extended if the put pointer is currently at the end of the file. If this pointer points into the middle of the file, characters in the file are overwritten with the new data.

If any error has occurred during writing in the file, the stream is placed in an error state.

Syntax of write method

```
ostream& write(const char*, int);
```

Reading

To read a binary file in C++ use read method. It extracts a given number of bytes from the given stream and place them into the memory, pointed to by the first parameter. If any error is occurred during reading in the file, the stream is placed in an error state, all future read operation will be failed then.

gcount() can be used to count the number of characters has already read. Then clear() can be used to reset the stream to a usable state.

Syntax of read method

```
ifstream& write(const char*, int);
```

```
#include<iostream>
#include<fstream>
using namespace std;
struct Student {
  int roll_no;
  string name;
};
int main() {
  ofstream wf("student.dat", ios::out | ios::binary);
  if(!wf) {
    cout << "Cannot open file!" << endl;
    return 1;
  }
  Student wstu[3];
  wstu[0].roll_no = 1;
  wstu[0].name = "Ram";
  wstu[1].roll_no = 2;
  wstu[1].name = "Shyam";
  wstu[2].roll_no = 3;
  wstu[2].name = "Madhu";
  for(int i = 0; i < 3; i++)
    wf.write((char *) &wstu[i], sizeof(Student));
  wf.close();
  if(!wf.good()) {
    cout << "Error occurred at writing time!" << endl;
    return 1;
  }
  ifstream rf("student.dat", ios::out | ios::binary);
  if(!rf) {
    cout << "Cannot open file!" << endl;
    return 1;
  }
  Student rstu[3];
  for(int i = 0; i < 3; i++)
```

```
    rf.read((char *) &rstu[i], sizeof(Student));
  rf.close();
  if(!rf.good()) {
    cout << "Error occurred at reading time!" << endl;
    return 1;
  }
  cout<<"Student's Details:"<<endl;
  for(int i=0; i < 3; i++) {
    cout << "Roll No: " << wstu[i].roll_no << endl;
    cout << "Name: " << wstu[i].name << endl;
    cout << endl;
  }
  return 0;
}
```

Output
Student's Details:
Roll No: 1
Name: Ram
Roll No: 2
Name: Shyam
Roll No: 3
Name: Madhu

Text file streams are those where the ios::binary flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function clear() can be used to reset the state flags.

get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:

ifstream, like istream, keeps an internal *get position* with the location of the element to be read in the next input operation.

ofstream, like ostream, keeps an internal *put position* with the location where the next element has to be written.

Finally, fstream, keeps both, the *get* and the *put position*, like iostream.

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

tellg() and tellp()

These two member functions with no parameters return a value of the member type streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case of tellp).

seekg() and seekp()

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:

seekg (position);
seekp (position);

Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.

The other form for these functions is:

seekg (offset, direction);
seekp (offset, direction);

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin",
ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << "
bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

❖ ASCII Files

An **ASCII File** is a **file** that contains unformatted **ASCII** text: only characters, numbers, punctuation, tabs, and carriage return characters. You can create and edit an **ASCII file** using Microsoft Notepad. ... txt, it is usually referred to as a text **file**, but you can save it with other extensions such as . bat

❖ Random Access operation

Random-access file is a term used to describe a **file** or set of **files** that are accessed directly instead of requiring that other **files** be read first. Computer hard drives **access files** directly, where tape drives commonly **access files** sequentially. Direct **access**, Hardware terms, Sequential **file**.

Here we will discuss how to access files randomly, forward and backward. Before moving forward or backward within a file, one important factor is the current position inside the file. Therefore, we must understand that there is a concept of file position (or position inside a file) i.e. a pointer into the file. While reading from and writing into a file, we should be very clear from where (which location inside the file) our process of reading or writing will start. To determine this file pointer position inside a file, we have two functions tellg() and tellp().

Position in a File

Let's say we have opened a file stream myfile for reading (getting), myfile.tellg () gives us the current get position of the file pointer. It returns a whole number of type long, which is the position of the next character to be read from that file. Similarly, tellp () function is used to

determine the next position to write a character while writing into a file. It also returns a long number.

For example, given an fstream object aFile:

```
Streampos original = aFile.tellp(); //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x; //write a value to file
aFile.seekp(original); //return to original position
```

So tellg () and tellp () are the two very useful functions while reading from or writing into the files at some certain positions.

Setting the Position

The next thing to learn is how can we position into a file or in other words how can we move forward and backward within a file. Suppose we want to open a file and start reading from 100th character. For this, we use seekg () and seekp () functions. Here seekg () takes us to a certain position to start reading from while seekp () leads to a position to write into. These functions seekg () and seekp () requires an argument of type long to let them how many bytes to move forward or backward. Whether we want to move from the beginning of a file, current position or the end of the file, this move forward or backward operation, is always relative to some position.. From the end of the file, we can only move in the backward direction. By using positive value, we tell these functions to move in the forward direction .Likewise, we intend to move in the backward direction by providing a negative number. By writing:

```
aFile. seekg (10L, ios::beg)
```

We are asking to move 10 bytes forward from the begining of the file. Similarly, by writing:

```
aFile. seekg (20L, ios::cur)
```

We are moving 20 bytes in the forward direction starting from the current position. Remember,

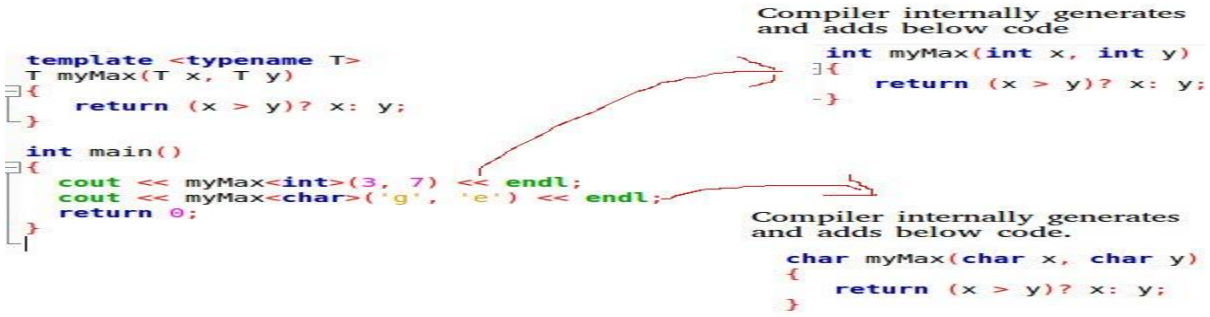
the current position can be obtained using the tellg () function.By writing:

aFile. seekg (-10L, ios:cur)

The file pointer will move 10 bytes in the backward direction from the current position. With seekg (-100L, ios::end), we are moving in the backward direction by 100 bytes starting from the end of the file. We can only move in the forward direction from the beginning of the file and backward from the end of the file.

❖ Templates

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.
C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.
Templates are expanded at compiler time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



Function Templates We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().
Know more on Generics in C++ .

```
#include <iostream>
```

```
using namespace std;

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
  return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```

Output:
7
7
g

❖ Exception handling

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Following are main advantages of exception handling over traditional error handling.

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Exception Handling in C++
Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

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

int main()
{
  int x = -1;

  // Some code
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:
Before try

Inside try
Exception Caught
After catch (Will be executed)

String, Declaring and initializing string objects, String attributes and Miscellaneous functions.

C++ has in its definition a way to represent sequence of characters as an object of class. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing access to single byte character.

Creating and initializing C++ strings
Create an empty string and defer initializing it with character data.
Initialize a string by passing a literal, quoted character array as an argument to the constructor.
Initialize a string using the equal sign (=).
Use one string to initialize another.
Use a portion of either a C char array or a C++ string.

std:: string vs Character Array
A character array is simply an array of characters can terminated by a null character. A string is a class which defines objects that be represented as stream of characters.
Size of the character array has to allocated statically, more memory cannot be allocated at run time if required. Unused allocated memory is wasted in case of character array. In case of strings, memory is allocated dynamically. More memory can be allocated at run time on demand. As no memory is preallocated, no memory is wasted.
There is a threat of array decay in case of character array. As strings are represented as objects, no array decay occurs.
Implementation of character array is faster than std:: string. Strings are slower when compared to implementation than character array.
Character array do not offer much inbuilt functions to manipulate strings. String class defines a number of functionalities which allow manifold operations on strings.

Operations on strings

Input Functions
1. getline() :- This function is used to store a stream of characters as entered by the user in the object memory.
2. push_back() :- This function is used to input a character at the end of the string.
3. pop_back() :- Introduced from C++11(for strings), this function is used to delete the last character from the string.

```
// C++ code to demonstrate the working of
// getline(), push_back() and pop_back()
#include<iostream>
#include<string> // for string class
using namespace std;
```

```
int main()
{
    // Declaring string
    string str;

    // Taking string input using getline()
    // "geeksforgeek" in giving output
    getline(cin,str);

    // Displaying string
    cout << "The initial string is : ";
    cout << str << endl;

    // Using push_back() to insert a character
    // at end
    // pushes 's' in this case
    str.push_back('s');

    // Displaying string
    cout << "The string after push_back operation is : ";
    cout << str << endl;

    // Using pop_back() to delete a character
    // from end
    // pops 's' in this case
    str.pop_back();

    // Displaying string
    cout << "The string after pop_back operation is : ";
    cout << str << endl;

    return 0;

}
```

Input:

geeksforgeek
Output:

The initial string is : geeksforgeek
The string after push_back operation is : geeksforgeeks
The string after pop_back operation is : geeksforgeek

Capacity Functions

4. capacity() :- This function returns the capacity allocated to the string, which can be equal to or

more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
5. resize() :- This function changes the size of string, the size can be increased or decreased.
6.length():-This function finds the length of the string
7.shrink_to_fit() :- This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters have to be made.

```
// C++ code to demonstrate the working of
// capacity(), resize() and shrink_to_fit()
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
// Initializing string
string str = "geeksforgeeks is for geeks";

// Displaying string
cout << "The initial string is : ";
cout << str << endl;

// Resizing string using resize()
str.resize(13);

// Displaying string
cout << "The string after resize operation is : ";
cout << str << endl;

// Displaying capacity of string
cout << "The capacity of string is : ";
cout << str.capacity() << endl;

//Displaying length of the string
cout<<"The length of the string is :"<<str.length()<<endl;

// Decreasing the capacity of string
// using shrink_to_fit()
str.shrink_to_fit();

// Displaying string
cout << "The new capacity after shrinking is : ";
cout << str.capacity() << endl;

return 0;
```

```
}
```

Output:

```
The initial string is : geeksforgeeks is for geeks
The string after resize operation is : geeksforgeeks
The capacity of string is : 26
The length of the string is : 13
The new capacity after shrinking is : 13
```

Iterator Functions

8. begin() :- This function returns an iterator to beginning of the string.
9. end() :- This function returns an iterator to end of the string.
10. rbegin() :- This function returns a reverse iterator pointing at the end of string.
11. rend() :- This function returns a reverse iterator pointing at beginning of string.

```
// C++ code to demonstrate the working of
// begin(), end(), rbegin(), rend()
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
    // Initializing string`
    string str = "geeksforgeeks";

    // Declaring iterator
    std::string::iterator it;

    // Declaring reverse iterator
    std::string::reverse_iterator it1;

    // Displaying string
    cout << "The string using forward iterators is : ";
    for (it=str.begin(); it!=str.end(); it++)
    cout << *it;
    cout << endl;

    // Displaying reverse string
    cout << "The reverse string using reverse iterators is : ";
    for (it1=str.rbegin(); it1!=str.rend(); it1++)
    cout << *it1;
    cout << endl;

    return 0;
```

```
}
```

Manipulating Functions

12. copy("char array", len, pos) :- This function copies the substring in target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied and starting position in string to start copying.

13. swap() :- This function swaps one string with other.

```
// C++ code to demonstrate the working of
// copy() and swap()
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
    // Initializing 1st string
    string str1 = "geeksforgeeks is for geeks";

    // Declaring 2nd string
    string str2 = "geeksforgeeks rocks";

    // Declaring character array
    char ch[80];

    // using copy() to copy elements into char array
    // copies "geeksforgeeks"
    str1.copy(ch,13,0);

    // Diplaying char array
    cout << "The new copied character array is : ";
    cout << ch << endl << endl;

    // Displaying strings before swapping
    cout << "The 1st string before swapping is : ";
    cout << str1 << endl;
    cout << "The 2nd string before swapping is : ";
    cout << str2 << endl;

    // using swap() to swap string content
```

```
    str1.swap(str2);

    // Displaying strings after swapping
    cout << "The 1st string after swapping is : ";
    cout << str1 << endl;
    cout << "The 2nd string after swapping is : ";
    cout << str2 << endl;

    return 0;

}
```

Output:

```
The new copied character array is : geeksforgeeks

The 1st string before swapping is : geeksforgeeks is for geeks
The 2nd string before swapping is : geeksforgeeks rocks
The 1st string after swapping is : geeksforgeeks rocks
The 2nd string after swapping is : geeksforgeeks is for geeks
```

*****