OBJECT ORIENTED PROGRAMMING WITH C++

UNIT-I

Object-Oriented Programming: Principles – Benefits of OOP – Application of OOP – Tokens, Expression and Control Structures: Tokens – Keywords – Identifiers and Constants – Data types – Constants – Variables – Operators – Manipulators – Expressions – Control Structure.

PRINCIPLES OF OBJECT ORIENTED PROGRAMMING:

Object oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful concepts.

It is defined as an approach that provides away of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

An object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data.

The concepts used extensively in object – oriented programming are

- **Objects**
- **Classes**
- **Data abstraction**
- **Inheritance**
- **Polymorphism**
- **Dynamic binding**
- **Message passing**

Objects:

Objects are the run-time entities in an object – oriented system.

They may represent a person, place, a bank account, a table of data or any item that the program has to handle.

When a program is executed, the objects interact by sending messages to one another. For example if “customer’’ and “account’’ are two objects in a program, then the customer may send a message to the account object requesting for the bank balance.

Objects can interact without having to know details of each other’s data or code. Objects contain code and data to manipulate that data.

Classes:

The entire set of data and code of an object can be made a user defined data type with the help of a class. Once a class has been defined, we can create any number of objects belonging to that class.

Each object is associated with the data of type class with which they are created.

A class is thus a collection of objects of similar type.

For example, mango, apple, and orange are members of class fruit.

Classes are user defined data types and behave like the built in types of a programming language.

Fruit mango;

will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation:

The wrapping up of data and functions into a single unit is known as Encapsulation. The data is not accessible to the outside world, and only to those functions, which are wrapped in the class, can access it.

These functions provide a interface between the object's data and the program.

This insulation of the data from direct access by the program is called data hiding or information hiding.

Abstraction refers to the act of representing essential features without including the background details or explanations.

They encapsulate all the essential properties of the objects that are to be created. These attributes are sometimes called data members because they hold information.

The functions that operate on these data are sometimes called methods or member functions.

Since the classes use the concept of data abstraction, they are known as Abstract Data Type(ADT).

Inheritance:

Inheritance is the process by which objects of one class acquire the properties of objects of another class .It supports the concept of hierarchical classification.

For example, the bird 'robin' is a part of the class 'flying bird', which is again a part of the class bird.

In OOP, the concept of inheritance provides the idea of reusability.

This means that we can add additional features to an existing class without modifying it.

This is possible by deriving a new class from the existing one.

The new class will have the combine features of both the classes.

$$\begin{tikzcd}
& \textbf{Bird} \arrow[dl] \arrow[dr] & & \\
\textbf{Flying bird} \arrow[d] \arrow[dr] & & \textbf{Non flying bird} \arrow[d] \arrow[dr] & \\
\textbf{Robin} & \textbf{Swallow} & \textbf{Penguin} & \textbf{kiwi}
\end{tikzcd}$$

<u>Polymorphism</u>:

Polymorphism is another important OOP concept.

Polymorphism means the ability to take more than one form .An operation may exhibit different behavior in different instances.

The behavior depends upon the types of data used in the operation .For example consider the operation of addition .For two numbers ,The operation will generate a sum .If the operands are strings ,then the operation would produce a third string by concatenation .

The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

<u>Dynamic binding</u>:

Binding refers to the linking of a procedure call to the code to be executed in response to the call.

Dynamic binding means that the code associated with a given procedure call is not known until the time of call at runtime .It is associated with polymorphism and inheritance. It is otherwise known as late binding.

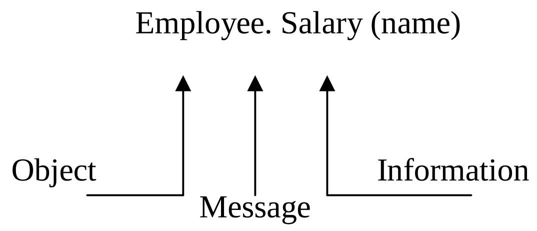
<u>Message Passing</u>:

An Object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object – oriented language, therefore, involves the following basic steps:

1. **Creating classes that define objects and their behavior**
2. **Creating objects from class definitions, and**
3. **Establishing communication among objects.**

Objects communicate with one another by sending and receiving information much the way people pass messages to one another.

Message Passing involves specifying the name of the object, the name of the function (message) and information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is possible as long as it is alive.

<u>BENEFITS OF OOP:</u>

OOP offers several benefits to both the designer and the user.

Object orientation contributes to the solution of many problems associated with the development and quality of software products.

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate With one another.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co exist without any interference
- It is possible to map objects in the problem domain to those in the program
- It is easy to partition the work in a project based on objects

- The data –centered design approach enables us to capture more details of a model in implementable form
- Object oriented designs can be easily upgraded from small to large systems
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

OBJECT -ORIENTED LANGUAGES:

The languages should support several of the OOP concepts to claim that they are object oriented.

Depending upon the features they support, they can be classified into the following two categories.

1. **Object-based programming languages, and**
2. **Object-oriented programming languages**

Object–based programming is the style of programming that primarily supports encapsulation and object entity. Major features that are required for object based programming are:

- **Data encapsulation**
- **Data hiding and access mechanism**
- **Automatic initialization and clear- up of objects**
- **Operator overloading**

Object – oriented programming incorporates all of object-based programming features along with two additional features, namely inheritance and dynamic binding.

Object oriented programming can therefore be characterized by the following statement:

Object-based features + inheritance + dynamic binding

APPLICATIONS OF OOP:

Applications of OOP are beginning to gain importance in many areas.

The most popular application of object oriented programming is in the area of user interface design .The promising areas of OOP include.

- ❖ **Real-time systems**
- ❖ **Simulation and modeling**
- ❖ **Object oriented database**

- **Hypertext, Hypermedia and expertext`**
- **AI and expert system**
- **Neural Networks and parallel programming**
- **Decision support and parallel programming**
- **CIM/CAM/CAD systems**

TOKENS, EXPRESSIONS AND CONTROL STRUCTURES:

C++ is a superset of c and therefore most constructs of c are legal in C++ with their meaning unchanged.

TOKENS:

The **smallest individual units in a program are known as tokens**. C++ has the following tokens:

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

KEYWORDS:

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user defined elements.

Given below are some of the keywords:

Private,do,protected,static,struct,volatile,while,virtual,void,friend,for,extern Auto,else,enum, etc.

IDENTIFIERS ANS CONSTANTS:

Identifiers refer to the names of variables, functions, arrays; classes, etc. create by the programmer. They are the fundamental requirement of any language.
Each language has its own rules for naming these identifiers.

The following rules are common to both C and C++

- **Only alphabetic characters, digits and underscores are permitted**
- **The name cannot start with a digit**
- **Uppercase and lowercase letters are distinct**
- **A declared keyword cannot be used as a variable name**

A major difference between C and C++ is the limit on the length of a name.

While C recognizes only the first 32 characters in a name C++ places no limit on its length
Constants refer to fixed values that do not change during the execution of a program. C++ supports several kinds of **literal constants**. They include integers, characters, floating point numbers and strings. **They do not have memory locations.**

Eg : 123 – decimal integer

12.34 - floating point integer

037 - Octal integer

0x2 – hexadecimal integer

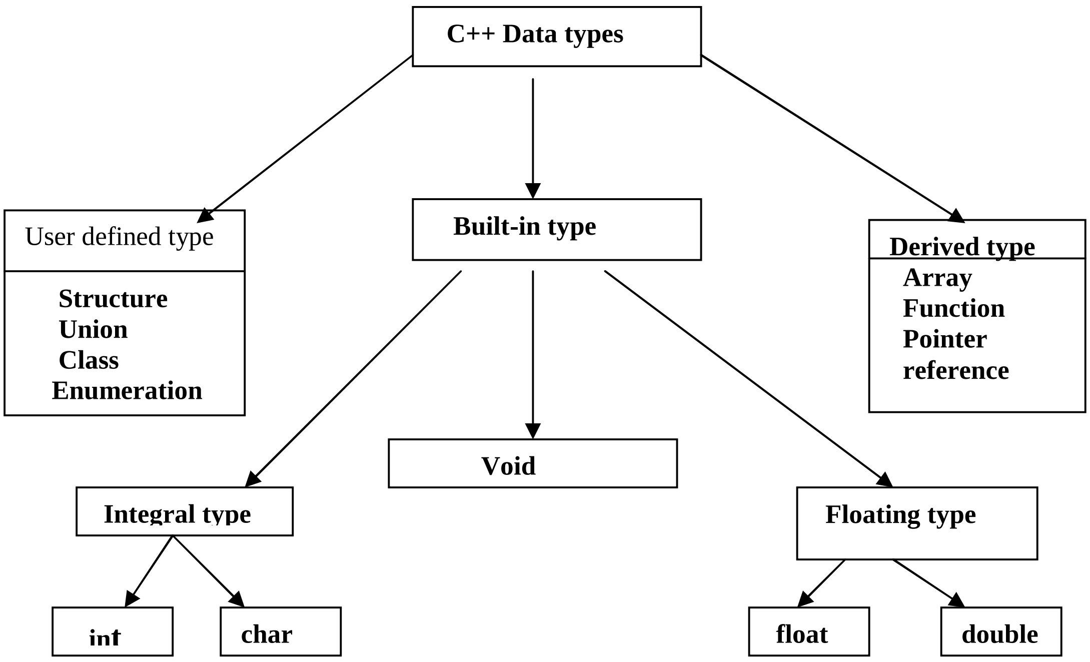
"C++"- string constant

'A' –character constant

C++ alaso supports backslash character constants such as'/a','/n','/t'.etc as in the case of C.

BASIC DATA TYPES:

Data types in C++ can be classified as follows:



C++ compilers support all the built in data types. they are also known as basic or fundamental data types .The modifiers signed,unsigned ,long,short may be applied to character and integer basic data types.

The modifier llong may also applied to double.
The type **Void** was introduced in C.
Two normal uses of void are

(1) to specify the return type of a function when it is not returning any value, and
(2) to indicate an empty argument list to a function

Eg. **Void funct1(void);**

Another use of **Void** is in the declaration of pointers

Eg .**Void *gp**

USER –DEFINED DATA TYPES:

Structures and classes:

C **++** also permits us to define user-defined data type known as class which can be used ,just like any other basic data type,to declare variables.

Enumerated Data Type:

An enumerated data type is another user-defined type which provides a way for attaching names to numbers.

The enum keyword automatically enumerates a list of words by assigning them values 0,1,2 and so on.

Enum shape{circle,square,triangle};
Enum colour{red,blue,green,yellow}

By using these tag names, We can declare new variables

Shape ellipse;
Colour background;

```
Eg: enum shape
       {
         circle;
         rectangle;
         triangle;
       };
       int main()
      {
      cout<< " enter shape code :";
      int code;
      cin>> code;
      while(code >= circle && code<= triangle)
      {
       switch(code)
      {
            case circle:
             ………
             ………
             break;
            case rectangle:
            ………
            ………
            break;
            case triangle:
            ………
            ………
            break;
       }
      cout << " enter shape code:';
      cin >> code;
        }
     cout<< "BYE \n";
      }
```

<u>DERIVED DATA TYPES</u>:

<u>Arrays</u>:

The application of arrays in C++ is similar to that in C. When initializing a character array in C,the complier will allow us to declare the array size as the exact length of the string constant.

```
        Char string[3] = "xyz";
```

But in C++,the size should be one larger than the number of characters in the string.

```
          Char string[4] ="xyz"
```

<u>Functions</u>:

Functions have undergone major changes in C++.While some of the changes are simple ,others require a new way of thinking when organizing our programs.

Many of the modifications and improvements were driven by the requirements of the object oriented concept of C++

<u>Pointers</u>:

Pointers are declared and initialized as in C

```
     Int *ip;
     ip =&x;
    *ip = 10;
```

C ++ adds the concept of constant pointer and pointer to a constant

```
   Char * const ptr1 ="GOOD";
```

<u>SYMBOLIC CONSTANTS</u>:

There are two ways of creating symbolic constants in C++

- Using the qualifier Const, and
- Defining a set of integer constants using **enum keyword**

In C++,w can use **const** in a constant expression ,such as

```
Const int size = 10;
Char  name[size];
```

A **const** in C++ defaults to the internal linkage and therefore it is local to the ffile where it is declared.

To give a **const** value an external linkage so that it can be referenced from, another file, we must explicitly define it as an **Extern** in C++.

```
Extern const total= 100;
```

Another method of naming integer constants is by enumeration as under

```
Enum {x,y,z}
```

This defines X,Y,Z as integer constants with values 0,1,and 2 respectively,
This is equivalent to :

```
Const x = 0;
Const y =1;
Const z = 2;
```

TYPE COMPABILITY:

C ++ defines int,short int and long int as three different types.

They must be cast when their values are assigned to one another. Similarly, unsigned char char and signed char are considered as different types, although each has assize of one byte.

These restrictions in C++ are necessary to support function overloading when two functions with the same are distinguished using the type of function arguments.

DECLARATION OF VARIABLES:

C++ allows the declaration of a variable anywhere in the scope.

This means that a variable can be declared right at the place of its first use.

This makes the program much easier to write and reduce errors that may be caused by having to scan back to forth. It also makes the program easier to understand because the variables are declared in the context of their use.

```
Eg:   int main()
        {
        float x;
        float sum = 0;
        for( int i  =1;I<5; I++)
        {
              cin>> x;
              sum =sum + x;
         }
         float average;
        average = sum/(I-1);
        cout<< average;
        return 0;
}
```

DYNAMIC INITIALIZATION OF VARIABLES:

C++ ,however ,permits initialization of the variables at run time.
This is referred to as dynamic initialization .In C++, avariable can be initialized at run time using expressions at th place of declaration.

Eg: **float average;**
Average = sum/i;
Can be declared as
Float average = sum/+ i

REFERENCE VARIABLES:

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alternative name for a previously defined variable.

A reference variable is created as follows

```
Data –type &reference name = variable-name
```

Eg:

```
Float total = 100;
Float&sum= total;
```

total is afloat type variable already declared;sum is the alternative name declared to represent the variable total.
The statements

```
cout<< total; and
cout<< sum;
```

will print the value 100.

A reference variable must be initialized at the time of declaration.

This establishes the correspondence between the reference and the data object which it names. A major application for reference variable is passing arguments to functions.

Eg.

```
    Void f(int & x)
    {
           x = x + 10;
    }
    int main( )
    {
     int m = 10;
     f(m);
     ……
     ……
   }
```

when the function call f(m) is executed, the following initialization occurs :

int & x =m;

Thus x becomes an alternate of m after executing the statement

f(m);

OPERATORS IN C ++

C ++ has a rich set of operators. These are given below:

: :	**Scope resolution operator**
: : *	**Pointer-to-member declare**
-> *	**Pointer-to-member operator**
.*	**Pointer-to-member operator**
delete	**Memory release operator**
endl	**Line feed operator**
new	**Memory allocation operator**
setw	**Field width operator**

SCOPE RESOLUTION OPERATOR:

C ++ is also a block structured language. Blocks and scopes can be used in constructing programs. the scope of the variable extends from the point of declaration.

A variable declare inside a block is said to be local to that block. consider the following segment

```
……………
……………
{
 int x = 10;
  ………
  ………
 }
 ……….
 ……….
```

```
{
  int x = 1;
  ………..
  ………..
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block and vice versa.

C ++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form :

```
: : Variable-name
```

This operator allows access to the global version of a variable.

MEMBER DEREFENCING OPERATORS:

C ++ permits us to define a class containing various types of data and functions as members. C ++ permits us to access the class members through pointers.

The member differencing operators are given below

..* To declare a member of a class

*** To access a member using object name and a pointer to that member**

->* To access a member using a pointer to the object and a pointer to that member

MEMORY MANAGEMENT OPERATORS:

C ++ supports **malloc** and **calloc() functions to allocate** memory dynamically
At run time. It also defines two unary operators new and delete that performs the task of allocating and freeing the memory later in a better and easier way.

Since these operates manipulate memory on the free store ,they are also known as free store operators.

A object can be created by using new, and destroyed by using delete, as and when required.

A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete.

Thus, the lifetime of an object is directly under our control and it is related to block structure of the program.

The new operator can be used to create objects of any type .It takes the following general form:

Pointer-Variable = new data type

The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object.

The data- type may be any valid data type.

The pointer variable holds the address of the memory space allocated.

```
P= new int;
q = new float;
```

Where **p** is a pointer of type int and q is a pointer of type **float**.

We can also initialize the memory using the new operator.
This is done as follows:

Pointer-Variable = =new data type (value)

Here value specifies the initial value.

New can be used to create a memory space for any data type such as arrays, structures and classes.

The general form for one – dimensional array is

Pointer-Variable = =new data type (size)

Eg.int *p =new int[10];

When a data object is no longer needed, It is destroyed to release the memory space for reuse. The general form of its use is :

Delete pointer- variable

The pointer –variable is the pointer that points to a data object created with New.

Eg. Delete P;

Delete q;

If we want to free a dynamically allocated array, We must use the following form
Delete:

Delete[size] pointer- variable

The size specifies the number of elements in the array to be freed.
The new operator offers the following advantages over the function **malloc()**

1. **It automatically computes the size of the data-object. We need not use the operate sizeof.**
2. **It automatically returns the correct pointer type, so that there is no need to use a type cast.**
3. **It is possible to initialize the object while creating the memory space.**
4. **Like any other operator new and delete can be overloaded.**

MANIPULATORS:

Manipulators are operators that are used to format the data display. The most commonly used operators are **endl** and **setw.**

The **endl** manipulator, when used in an output statement causes a linefeed to be inserted. It has the same effect as using the newline character "\n".

```
Eg.Cout << "m =" << m << endl
     Cout << "n=" << n<< endl
     Cout << "p =" << p << endl
```

If we assume the values of the variables 2597,14 and 175 the output will appear as follows

M=

2	5	9	7

N=

1	4

P=

1	7	5

The mainpualtor **setw (5**) specifies a field width for printing the value of the variable sum.

		3	4	5

TYPECAST OPERATOR:

C++ permits explicit type conversion of variables or expressions using the type cast operator.
The following two versions are equivalent
(type-name) expression
type-name(expression)

```
Eg.  Average = sum/(float) i ;
     Average = sum/(float) i
```

C ++ also supports the following new cast operators:

- **Const_cast**
- **Static_cast**
- **Dynamic_cast**
- **Reinterpret_cast.**

EXPRESSIONS AND THEIR TYPES

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function call which returns values.

An expression may consist of one or more operands and zero or more operators to produce a value.

Expressions may one of the following seven types

- **Constant expressions**
- **Integral expressions**
- **Float expressions**
- **Pointer expressions**
- **Relational expressions**
- **Logical expressions**
- **Bitwise expressions**

Constant expressions:

Constant expressions consist of only constant values.
Eg 15 , 20 + 5 / 2.0 ,'x'

Integral expressions:

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
E. m,m*n-5,m*'x'
Where m and n integers

Float expressions:

Float expressions are those which, after all conversions produce floating point results
Eg.X+Y,5+Float (10)
Where X and Y are floating point variables

<u>Pointer Expressions</u>

Pointer expressions produce address values
Eg. &m,ptr
Wher m= is a variable and ptr isa pointer

<u>Relational Expressions</u>

Relational expressions yield results of type **Bool** which takes a value true or false
  Eg.X <= y
       a+b = = c +d
When a arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results are compared. Relational expressions are also known as Boolean expressions,

<u>Logical Expressions</u>:

Logical expression combines two or more relational expressions and produces **BOOL** Results.
Eg: a > b && x = =10

<u>Bitwise Expressions</u>:

Bitwise expressisons are used to manipulate data at bit level. They are basically used for testing or shifting bits
 X << 3; Shifts three bit position to left
 Y >> 1;Shifts one bit position to right;

<u>SPECIAL ASSIGNMENT OPERATOR</u>:

There are three type of assignment operators:

1) **Chained assignment :**

   X = (y = 10)

   First the value is assigned to x then to y.
   A chained assignment cannot be used to initialize variables at the time of decalaration.

2) **Embedded assignment:**

   X = (y= 50) + 10;

   Here ,the value 50 is assigned to y and then the result is assigned to x. This statement is identical to

   Y = 50;
   X = y +10;

3) **Compound assignment:**

C ++ supports a compound assignment operator which is a combination of the assignment operator with a binary operator. For example

X = X + 10;

May be written as

X += 10;

The operator += is known as assignment operator or short hand assignment operator. The general form is

Variable1 op = variable 2;

Where op is a binary operator

IMPLICIT CONVERSION:

Wherever data types art mixed in an expression, C++ performs the conversions automatically. This process is known as implicit or automatic conversion.

OPERATOR OVERLOADING:

C ++ permits overloading of operators, thus allowing us to assign multiple meaning to operators. The input /output operators << and >> are good examples of operator overloading.

This has been made possible by the header file iostream where a number of overloading definitions are included.

OPERATOR PRECEDENCE:

C ++ enables us you add multiple meanings to the operators, yet their association and precedence remain the same.

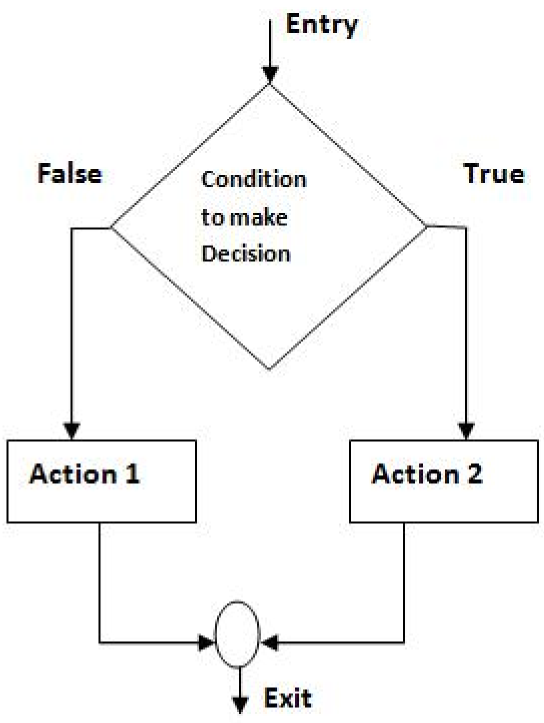
Operator	Associativity
::	Left to right
->,( ), [ ],postfix ++,postfix - - , prefix ++,prefix- - ,~ !	Left to right
-> * *	Right to left
* ,/,%	Left to right
+ -	Left to right
<<,>>	Left to right
<< =,>> =	Left to right
= = ! =	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
? :	Left to right
=* ,= /, =%, =+,= =	Left to right
<< =,>> =,&=,^= ,| =	Right to left
,	Left to right

CONTROL STRUCTURES:

There are three types of control structures:

1. **Sequence structure(straight line)**
2. **Selection structure (Branching)**
3. **Loop structure (iteration)**

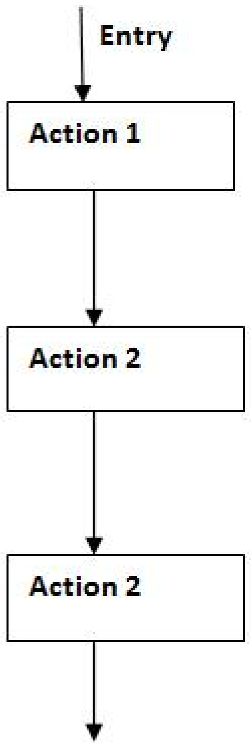
The following figure illustrates this :



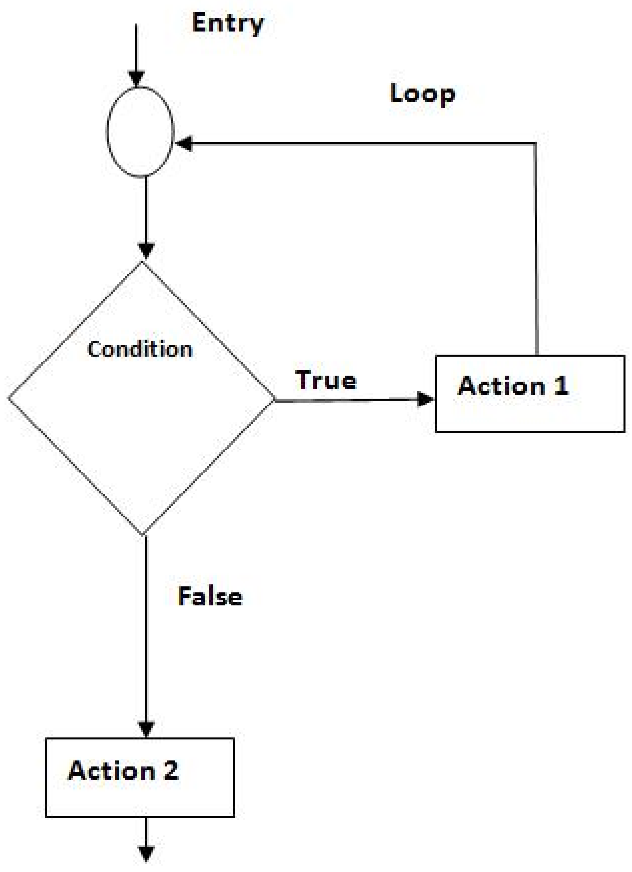
Selection Structure

Implemented using:- If and If..else control statements

switch is used for multi branching



Sequence Structure



Loop Structure

Implemented using:- While , Do While and For control statements

---

The If statement:

The **If** statement is implemented in two forms :

- Simple **if** statement
- **If …else** statement

Form 1

```
 If ( expression is true)
{
    action1;
}
}
action2;
action3;
```

Form2 :

```
If ( expression is true )
{
        Action 1;
}
else
{
        action 2;
}
action3;
```

Example program for if statement in C:

In “if” control statement, respective block of code is executed when condition is true.

```
int main()
{
  int m=40,n=40;
  if (m == n)
  {
      cout<<"m and n are equal";
  }
}
```

Output:

```
m and n are equal
```

Example program for if else statement in C:

In C if else control statement, group of statements are executed when condition is true. If condition is false, then else part statements are executed.

```
#include <iostream.h>
int main()
{
  int m=40,n=20;
  if (m == n) {
      cout<<"m and n are equal”;
  }
  else {
        cout<<"m and n are not equal";
  }

}
```

Output:

```
m and n are not equal
```

Example program for nested if statement in C:

- In “nested if” control statement, if condition 1 is false, then condition 2 is checked and statements are executed if it is true.
- If condition 2 also gets failure, then else part is executed.

```
#include <iostream.h>
int main()
{
  int m=40,n=20;
  if (m>n) {
      cout<<"m is greater than n";
  }
  else if(m<n) {
        cout<<"m is less than n";
  }
  else {
        cout<<"m is equal to n";
  }
}
```

Output:

```
m is greater than n
```

The switch statement:

This is a multiple – branching statement where, base dona condition, the control transferred to one of the many possible ways

```
Switch(Expression)
{
   case 1:
    {
      Action 1;
    }
   case 2 :
    {
      Action 2;
    }
   case 3:
    {
      Action 3;
    }
   default :
    {
      Action 4;
    }
}       action 5;
```

The do-While statement:

The do-While is an exit-controlled loop. Based on condition, the control is transferred back to the particular point in the program. The syntax is as follows:

```
   Do
 {
    action1;
  }
 while (condition is true );
 action2;
```

While statement:

This is also a loop structure, but is an **entry-controlled one.** The syntax is as follows:

```
while (condition is true );
{
   action1;
 }

action2;
```

The for statement:

The **for** statement is an entry controlled and is used when action is to be repeated for a predetermined number of times. The syntax is as follows:

```
 For( initial value;test;increment)
{
   action1;
 }

action2;
```

Example program (for loop) in C++:

In for loop control statement, loop is executed until condition becomes false.

```
#include <iostream.h>

int main()
{
  int i;

  for(i=0;i<10;i++)
  {
     cout<<i;
  }
}
```
Output:

```
0 1 2 3 4 5 6 7 8 9
```

Example program (while loop) in C++:

In while loop control statement, loop is executed until condition becomes false.

```
#include <iostream.h>

int main()
{
  int i=3;

  while(i<10)
  {
     cout<<i;
     i++;
  }

}
```

Output:

```
3 4 5 6 7 8 9
```

Example program (do while loop) in C++:

In do..while loop control statement, while loop is executed irrespective of the condition for first time. Then 2nd time onwards, loop is executed until condition becomes false.

```
#include <iostream.h>

int main()
{
  int i=1;

  do
  {
     cout<<"Value of I is:"<<i;
     i++;
  }while(i<=4 && i>=2);

}Output:
```

```
Value of i is 1
Value of i is 2
Value of i is 3
Value of i is 4
```

Difference between while & do while loops in C++:

S.no	**while**	**do while**

1	Loop is executed only when condition is true.	Loop is executed for first time irrespective of the condition. After executing while loop for first time, then condition is checked.

UNIT –II

Functions: Prototyping - Call by Reference - Return by Reference - Inline Functions - Default Arguments - const Arguments - Function Overloading - Friend and Virtual Functions, Classes and Objects - Class - Member Functions - Arrays with in a Class - Memory Allocation for Objects - Static data members - Static member functions - Arrays of Objects - Objects as Function Arguments - Friendly Functions - Returning Objects - const Member Functions - Pointers to Members, Constructors and Destructors.

Functions in C++:

Dividing a program into functions is one of the major principles of top-down structured programming.

Advantages:

- Reduce the size of the program
- Reusability of code.

Syntax:

```
 Void main ()
{
Declaration;
Statements:
Return (0);
}
```

```
Main()
{
        message(); ———→ calling function
        cout<<”hai”;
}
message()
{
        cout<<”hello”; ———→called fucntion
}
```

Actual parameters:

It is defined in calling function and they have actual values to be passed to the called function.

Formal parameters:

It is defined in called function and they receive values of actual parameters when function is invoked.

Function prototype:

The prototype describes the function interface to the compiler by giving details such as

- The number and type of arguments and
- The type of return values

Function prototype is a declaration statement in the calling program and is of the following form;

type function-name(argument-list);

The argument-list contains the types and names of arguments that must be passed to the function.

Ex: float volume(int x, float y, float z);

Function prototype describes the function interface

<u>Function definition:</u>

A function definition has a name, a parenthesis pair containing zero or more parameters and a function body. Any parameter not declared is taken to be int by default.

Syntax: function type function-name (data type argument1, data type argument2…..)

```
        {
          Body of function
          ---------
          ---------
        Return something
               }
```

Ex:

```
    Float volume(int a, float b, float c)
    {
      float v=a*b*c;
      ….}
```

Call by value:

- Called function creates new variables to store the value of arguments passed to it.
- It copies the value of actual parameters into the formal parameters.
- Thus the function creates its own copies of arguments and then uses them.

```
Void swap(int,int);
Void main()
{
int a,b;
cin>>a>>b;
cout<<"before swap";
swap(a,b);
cout<<a<<b;
}
void swap (int  a1, , int  b1 );
   {
   int temp;
   temp= a1;
   a1=b1;
   b1=temp;
   cout<<a<<b
   }
```

<u>Call By Re</u>

Call by reference is a method in which the address of each argument is passed to the function. By this method, the changes made to the parameters of the function will affect the variables in the calling function.

Ex

```
// call by reference
#include<iostream.h>
void swap (int *x, int *y );
{
int temp;
temp= *x;
x=*y;
*y=temp;
}
void main ( )
{
int x = 10,= 20
swap (&x, &y);
cout << x <<y;
}
```

Return By Reference:

A function can also return a reference

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

Since the return type of max() is int&, the function returns reference to x or y . then a function call such as max(a,b) will yield a reference to either a or b depending on their values.

Return Statement:

The keyword return is used to terminate function and return a value to its caller. It also be used to exit a function without returning any value. It is used in anywhere within function body.

Syntax: **return;**
return(expression);

Ex:

```
int sum(int x,int y)
{
 return(x+y);      }
float maximum(float b,float a)
{
   if(a>b)
      return(a);
   else
      return(b);
}
```

Inline Function:

An inline function is a function that is expanded inline when it is invoked. The compiler replaces the function called with the corresponding function code.

Syntax:

```
inline        function header.
{
function body;
}
```

Ex:

```
Inline float convert_ feet(int x)
{
        return x*12;
}
void main()
{
        int inch=45;
        cout<<convert-feet(inches);
}
```

Disadvantages of Inline function:

It makes the program to take more memory because the statements that define the inline function are reproduced at each point where the function called.

Default Arguments:

C++ allows us to call a function without specifying all its arguments. In such cases the function assigns a default value to the parameter, which does not have a matching argument in the function call. Default values are specified when the function is declared.

Ex:

```
 int sum (int a, int b)
 x = sum (10, 20);
passes the value of 10 to a and 20 to b.
```

Advantages

- ❖ Default argument helps to add new parameters to the existing functions
- ❖ Default argument can be used to combine similar functions into one

Constant Arguments:

An argument to function can be declared as constant

Eg: int strlen(const char *p);

int length (const , string&s);

The qualifier constant tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated.

Function overloading:

✦ Overloading refers to the use of the same thing for different purpose. This means that we can use the same function name to create functions that perform a variety of different tasks.

Function overloading:

We can design a family of function with one function but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

```
//Declaration
int add(int a,int b);
int add(int a,int b,int c);
double add(double x,double y);
double add(int p,double q);
double add(double p,int q);
//function call
cout << add(5,10);
cout<<add(15,10.0);
cout<<and(12.5,7.5);
cout<<add(5,10,15);
cout <<add(add(0.75,5);
```

A function call first matches the prototype having the same number and type of argument and then calls the appropriate function for execution .

Ex:

```
#include<iostream.h>
//declaration
int  volume(int);
double volume(double,int);
long volume(long ,int ,int);
int main( )
        {
        cout << volume(10) << "\n";
        cout<<volume(2.5,8) << "\n";
        cout << volume(100,75,15) << "\n";
        return 0 ;
}
// function definition
int volume(int s)   //cube
```

Friend Function:

Characteristics Of Friend Function:

- It is not in the scope of the class to which it has been declared as friend
- It cannot be called using the object of the class
- It can be invoked without the help of any object
- It cannot access the member names directly
- It has can be declared either private or the private part of a class
- It has objects as arguments

Virtual Functions:

A virtual function is one that does not really exists but it appears real in some parts of program.

Syntax:

```
Class class_name
 {      private:
           members;
        public :
        virtual return_type function_name();
  };
```

Ex:

```
Class student
          {     private:
                int rollno;
                float avg;
                public:
                virtual void getdata();
                virtual void display();
           };
```

When using the same function name in both the database and derived classes, the function in base class is declared as virtual using the key board virtual preceding its normal declaration.

When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to different objects we can execute different version of the virtual function.

Rules For Virtual Functions:

- Virtual function can also have inline code substitutions.
- The Virtual function must be members of some class.
- They cannot be static.
- They are accessed by using object pointers.

- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- If two functions with the same name have different prototype, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- We cannot have virtual constructors, but we can have virtual destructors.
- While a base pointer can point to any type of the derived object, the reverse is not true.(i.e) we cannot use a pointer to a derived class to access an object of the base type.
- When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure virtual function:

- It is a type of which has only a function declaration and does not have the function definition.
- In general, to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serve as a place holder.
- A ”do-nothing” function may be defined as follows;
    - **virtual void display ( ) = 0;**

- It is a function declared in a database class that has no definition relative to the base class.
- In such a cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

Ex:
```
class base
{
  int x;
  float y;
  public:
       virtual void getdata();
       virtual void display();
   };

class derived : public base
{       ---
        ---   };
void base :: getdata() // pure virtual function definition
{       ---
        ---  }
void base :: display() // pure virtual function definition
{       --- - }
```

Classes and Objects:

Specifying Classes:

A class is a way to bind the data and its associated functions together. When defining a class, we are creating a new abstract data type that can be treated like any other built-in-data types.

- **Class definition**
- **Class function declarations**

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The General Form of a class declaration is

```
Class class-name
{
 private:
        variable declarations;
        function  declarations;
  public:
        variable declarations;
        function declarations;
   protected:
        variable declarations;
        function declarations;
};
```

The class declaration is similar to a struct declaration. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called members. They are usually grouped under two sections.

(1) Private
(2) Public.

- The keywords private and public are known as **visibility labels.** The members have been declared as **private can be accessed only from within the class. The public members can be accessed outside the class**.
- The **data hiding** is the key features of object-oriented programming. The use of keyword private is optional. By default, the members of a class are private.
- The variables declared inside the class are known as **data members** and the functions are known as **member functions**. Only the member functions can have access to the private data members and private functions.
- However, the public members can be accessed fro outside of the class. The binding of data and functions together into a single class type, variable is referred to as encapsulation

Example:

```
Class item
{
  int no;
float cost;
public:
void get data(int a,int b)
void put data(void);
};
```

Object Of A Class:

A class is a user defined data type, while an object is an instance of class template. A class provides a template, which defines the member function and variables that are required for object of the class type. Once a class has been created, we can create any number of objects belonging to that class. We can also declare more than one object in one statement.

```
Ex: item x;   //  memory  for  x  is
created
       item x,y,z;
for example:
class item
{

……….
……… } x,y,z;
```

Accessing class members:

The private data of a class can be accessed only through the member functions. The object can access a variable declared as public directly.

Syntax:

object-name. Function-name (actual-arguments);

Ex: x.getdata (100,75.5);

Example:

```
#include<iostream>
class item
{
private:
        int number;
        float cost;
public:
       void getdata (int a, float b);             //declaration
       void putdata(void);     //declaration
void item :: putdata (void)
         {
         cout << "Number:" <<number <<endl;
         cout<< "Cost :" << cost << endl;
   }
};
void item :: getdata (int a , float b)
           {    number = a;
                 cost = b;
            }
         Int Main()
         {
         Item x;
         Cout<<"object x"<<"\n";
         x.getdata(100,299.95);
         x.putdata();
         item y;
         cout<<"object y"<<"\n";
         y.getdata(200,175.50);
         g.putdata();
         }
```

Defining member function:

Member functions can be defined in two places:

- **Outside the class definition**
- **Inside the class definition**

Outside the class definition:

Member function that are declared inside a class have to be defined separately outside the class. The member function is defined outside the class by using the following way:

Syntax:

```
return-type class-name :: function-name (argument declaration)
{       function body
 }
```

The membership label class-name :: tells the compiler that the function name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified in the header line.

The **symbol :: is called the scope resolution operator.**

Example:

```
Void item::getdata(int a, float b)
{
Number = a;
 Cost = b;
}
Void item :: put data(void)
{
Cout<<"number"<<number<<"\n";
Cout<<"cost:"<<cost<<"\n";
}
```

Inside the class definition:

Another method of defining a member function is to replace the function declaration by the function definition inside the class.

Syntax:

```
   return type function name(argument decl
aration)
                  {   function body;
                  }
```

Example:

```
class item

{
   private:
        Int number;
        float cost;
   public:
     void get data (int a, float b);          //declaration
```

```
        void put data(void);                    //definition
      {
            cout <<number<< “\n”;
           cout <<cost << “\n”;
      }
   };
```

Making an outside function inline:

We can define a member function outside the class definition and still make it inline by just using the qualified inline in the header line of function definition.

Eg:

```
Class item
{
   int number ,cost;
public :
void get data ( int a, float b);/* declaration*/
};

inline void item :: get data( int a, float b);    /*definition*/
{
number =a;
cost=b;
}
```

Nesting Of Member Function & Private Member Function:

Nesting Of Member Function:

A number function can be called by using its name inside another member function of the same class. This is known as nesting of member functions

Example program nesting of member function:

Class set
{
int m,n;
public:
void input(void);
void display(void);
int largest(void);
}
int set::largest(void)
{
if(m>=n)
	Void set::input(void)
	{
	 cout<<”enter m and n “;
	cin>>m>>n;
	}
	void set::display(void)
	{
	  cout<<”largest
	value”<<largest();
	}
	int main()

{
return(m);
else
return(n).
}
	{
	  set a;
	a.input();
	a.display();
	return 0;
	}

Private Member Functions:

A private member function can only be called by another function that is member of its class. Even an object cannot invoke a private function using dot operator.

Ex:

```
        Class sample
{           int m;
            void read();

  public:
          void update();
};
void sample: :update()
{
        read();
   }
```

<u>Arrays within a class:</u>

An array is a user data type whose member is homogeneous and stored in contiguous memory locations. The arrays can be used as member variable in a class.

Example:

```
class array
{      int a [size];
public:
      void setval (void);
     void display (void);
};
```

The array variable a [] declared as a private member of the class array can be used in the member function like any other variable. In the above class definition the member function setval( ) sets the values of elements of the array a [ ] and display( ) function displays the values. Similarly we may use other member function to perform any other operation on the array values.

```
Class fact
{
   int n, fact;
public:
    void get();
    void display();
};
 void fact::get()
  {
   cout<<"enter n";
   cin>>n;
   fact=1;
   for(int i=1;i<=n;i++)
   {
      fact=fact*I;
  }
}
```

```
Void fact::display()
{
  cout<<fact;
}
void main()
{
   fact n;
   n.get();
   n.display();
}
```

Memory Allocation Of Objects:

- The memory space for object is allocated and when they are declared and not when class is specified. Space for member variables is allocated separately for each objects .
- Separate memory locations for the objects are essential because the member variables will hold different data values for different objects.

Static Data Members:

Characteristics:

- It is declared within the class, but its lifetime is the entire program.
- It is initialized to zero when the first object of its class is created. No other initialization permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects that class, matter how many objects are created.
  It is visible only

```
Class item
{
 static int count;
int num;
public:
   void getdata(int a)
{
   number=a;
    count++;
}
void getcount(void)
{
 cout<<"Count";
 cout<<count;
}
};
```

```
Int item::count
Int main()
{
  item a,b,c;
  a.getcount();
  b.getcount();
  c.getcount();
  a.getdata(100);
  b.getdata(200);
  c.getdata(300);
  cout<<"after reading data";
  a.getcount()
  b.getcount();
  c.getcount();
 return 0;
}
```

.

<u>Static member functions:</u>

A member function that is declared as static has the following properties:
A static member function can have access to only other static members declared in the same class.
A static member function can be called using the class name [instead of its objects].

class-name::function-name;]

<u>Arrays Of Objects:</u>

An array can be of any data type including struct. Similarly we can also have arrays of variables that are of the type class. Such variables are called arrays of objects.

Ex:

```
class emp
{       char name [30]
        float age;
public:
       void getdata (void);
       void putdata (void);
```

The identifier emp is a user-defined data type & can be used to create objects that related to different categories of the employees.

Storage Of Data Items Of An Object Array

emp manager [3];
emp worker [10]'

The array manager contain 3 objects namely manager [0], manager [1] & manager [2], similarly worker contain 10 objects.

An array of objects behaves like any other array; we can use the usual accessing array methods to access individual elements and then the dot member operator to access the member function.

Ex:

manager[i] .put data();

The above will display the data of the ith element of the array manager.

Object As Function Arguments:

Object may be used as a function argument. This can be done in two ways

1. A copy of the entire object is passed to the function.
2. Only the address of the object is transferred to the function.

- The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.
- The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.
- The pass-by-reference method is more efficient since it requires passing only the address of the object and not the entire object.

Friendly functions:

A friend is a function that is not a member of a class but has access to the private members of the class. Normally private members are hidden from all parts of the program outside the class and accessing them requires a call to public member function. To make an outside function friendly to a class , we have to simply declare this function as a friend of the class as shown below:

```
Class ABC
{   --------
    --------
public:
   --------
    --------
friend void xyz(void) // declaration
};
```

- The function declaration should be preceded by the keyword **friend.**
- The function is defined elsewhere in the program like a normal C++ function.
- The function definition does not use either the keyword **friend** or the **scope operator ::.**
- The function that are declared with the keyword friend are known as friend function. A function can be declared as a friend in any number of classes.
- A friend function ,although not a member function , has full access rights to the private members of the class.

Characteristics:

- It is not in the scope of the class to which it has been declared as friend
- It cannot be called using the object of the class
- It can be invoked without the help of any object
- It cannot access the member names directly
- It has can be declared either private or the private part of a class
- It has objects as arguments
- It has often used in operator overloading

Example:

```
Class sample
{
   int a;
   int b;
   public:
     void setvalue()
       {a=25;
        b=40;
        }
     friend float mean(sample s);
  };
```

```
float mean(sample s)
{
           return float(s.a+.s.b)/2.0;
        }
        void main()
        {
          sample x;
            x.setvalue();
      cout<<"mean
value="<<mean(x)<<"\n";
    return 0;
        }
```

Returning object:

A function cannot only receive objects as arguments but also can return them .In the below example we can create object and how object is return to another function is shown below

Example:

```
#include<iostream.h>
Class   complex
{
float x;
float y;
public:
```

```
void complex :: show ( complex c)
{
cout<< c.x<<"+j"<<c.y<<"\n";
}
int main()
{
```

```
void input ( float real, float imag)
{
x= real;
y= imag;
}
friend complex sum( complex, complex);
void show ( complex);
};
complex sum ( complex c1, complex c2)
{
complex c3;
c3.x= c1.x+c2.x;
c3.y = c1.y+c2.y
return (c3) ;
}
```

```
complex A,B,C;
A. input(3.1,5.65);
B.input( 2.1,1.2);
C= sum (A,B);    ///C=A+B
cout<< " A= ";  A. show()
cout<< " B= ";  B.show()
cout<< " C= ";  C.show()
return 0;
}
```

output:

```
A=3.1+j5.65
B=2.75+j1.2
C=5.85+j6.85
```

<u>Const member function:</u>

- A const member function guarantees that it will never modify any of its class member data
- If a member function does not alter any data in the class, then we may declare it as **const** member function.

Eg: Void mul(int,int) const;

Double get balance() const;

- The qualifier **const** is appended to the function prototypes(in declaration and definition).
- The compiler will generate an error message if such functions try to alter the data values.

<u>Pointers to members:</u>

It is possible to take the address of a member of a class can be assigned it to a pointer. The address of a member can be obtained by applying the operator '&' to a fully qualified class member name. The class member pointer can be declared using the operator ::* with the class name

Ex:

```
Class A
{
Private:
Int m;
Public:
Void show();
};
```

<u>Local class:</u>

Classes can be defined and used inside a function or a block. Such classes are called as local classes.

Example:

```
Void test(int a)                    //  function
{
………


………
        class student               //  local class
{
……..
…….                                //  class definition
};
…….
Student s1(a);                   //   create student object
use student object
```

- Local classes can use global variables and static variables declared inside the function cannot use automatic local variable.
- The global variable should be used with the scope operator(::)
- There are some restrictions in constructing local classes**.** They cannot have static data members and Member functions must be defined inside the local classes.
- Enclosing function cannot access the private members of a local class. But it can done by declaring the enclosing function as a friend

Constructor And Destructor:

It is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created .It is called constructor because it construct the values of data members of the class.

Ex:

```
#include<iostream.h>
class add
{
private :
   int a,b,c;
public:
   add( ) ;    //constructor
      {
        a=10;
         b=20;

      }
```

```
  void display( );
     {
        cout << a+b<<"/n";
     }
};
void main( )
  {
  add d;
  d.display( );
  }
```

Constructors.

Characteristics:

- ★ They should be declared in the public section.
- ★ They are invoked automatically when the objects are created.
- ★ They do not have return types, not even **void** and therefore they cannot return values.
- ★ They can't be inherited, though a derived class can call the base class constructor.
- ★ They can have default arguments.
- ★ Constructor can't be virtual
- ★ We cannot refer to their addresses.
- ★ An object with a constructor can't be used as a member of a union.
- ★ They make implicit calls to the operators new and delete when a memory allocation is required.

- ★ There is no return type specified. A constructor many contain a return statement without an argument about any attempt to return any type of value from a constructor is an error.
- ★ The name of the constructor function is the same as name of the class.
- ★ An initialization list may follow the name & argument of the constructor. The list is a comma, separated list of class member with associated initial values. The list is separated from the argument list by a single colon.

Parameterized Constructor:

C++ permits us to achieve this objective by passing argument to the constructor function when the objects C++ are created. The constructor that can take argument are called parameterized constructor.

The constructor integer( ) may be modified to take argument as shown below

```
        class integer
        {
                int m.n;
        public:
                integer(int m, int y)             //parameterized
    constructor
                statement;
        };
        integer :: integer(int x , int y)
        {
        m=x;
        x=y;
        }
```

We must pass the initial values as argument to the constructor function when an object is declared. This can be done in 2 ways.

1.By calling constructor explicitly

2.By calling the constructor implicitly

Explicit constructor:

integer int1=integer(0,100)

This method creates an integer object in it and passes the values 0 and 100 to it

Implicit constructor:

integer int1(0,100)

In this method, sometimes called the shorthand method is used very often

Program for Multiple Constructors:

```
class integer
     {
            int m,n;
     public:
        integer ( )                 //constructor 1
          {
           m=0;
           n=0;
          }
        integer (int a, int b)       // constructor 2
         {
          m=a;
          n=b;
         }
integer (integer &  i)               // constructor 3
  {
   m=i.m;
   n=i.n;
   }
};
```

The first receives no argument. The 2nd receives two integer arguments & the 3rd receives one integer object as an argument.

Constructor With Default Arguments:

The constructor is also defined with default arguments that means, we can give the default value to the arguments at the time of declaration.

Example:

```
Class complex
        {
          private:
             int x;
          public:
             complex(float real, float imag=0);
        };
```

The default value of the argument imag is zero. Then, the statement complex c(5.0) . Assigns the value 5.0 to the real and 0.0 to imag(by default). However, the statement complex c(2.0,3.0) assigns 2.0 to real and 3.0 to imag respectively.

Dynamic Initialization Of Objects:

Class objects can be initialized dynamically too. That is, the initial value of an object can be provided during run time

Advantage:

Dynamic initialization is that we can provide various initialization formats, using overloaded constructors.

Copy Constructor:

- ★ Copy constructor is used to declare and initialize an object from another object.
- ★ It takes a reference to an object of the same class as itself as an argument.
- ★ Every class at least 2 constructors there are identified by their unique declaration.

Copy initialization:

- ✓ The process of initializing through a copy constructor is known as **copy initialization**.
- ✓ A copy constructor takes a reference to an object of the same class as itself as an argument. Consider a simple example of constructing and using a copy constructor

Ex:

```
class code
{
      int id;
      public:
        code();  {}// constructor
        code(int a);
          { id=a ;}
        code(code &x)
        {    id=x.id;
        }
        void display()
        {
             cout<<id;
         }
    };
```

```
void main()
{
        code a(100);
        code b(a);
         code C=A;
         code D;
         D=A;
        a.display();
         b.display();
         c.display();
         d.display();
         return 0;
 }
```

Dynamic constructor:

The constructor can also be used to allocate memory while creating object. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory at the

time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

Const objects:

We may create and use constant objects using const keywords before object declaration.

Example:

we may create X as a constant object of the class matrix as follows

Const matrix X(m,n); // object X is constant

- Any attempt to modify the values of m and n will generate compile time error. A constant object can call only const member function. Const member is a function prototype or function definition where the keyword const appears after the functions signature.
- Whenever const objects try to invoke non-const member functions, the compiler generates errors.

Destructors:

A destructor like a constructor has the same name as that of the class but is prefixed by the tilde (~) symbol. It is only invoke when the object is destroyed. Add a destructor can't take arguments a specify a return values.

Syntax:

~matrix ( );

A Destructor never takes any arguments not does return any value. Whenever new is used to allocate memory in the constructor we should use delete to free that memory.

Example:

```
matrix ::  matrix ( )
{    for  (i=0; i <n; i++)
     delete p [i];
       delete p;
 }
```

```
#include<iostream.h>
Int count =0;
Class alpha
  {
        Public:
        Alpha()
        {
        Count++;
        Cout<<"\n No.of object created " <<count;
        }
        ~alpha()
        {
        Cout<<"\n No.of object destroyed"<<count;
        Count--;
        }
```

UNIT-III

Operator Overloading and Type Conversions - Inheritance: Extending Classes - Derived Classes - Single Inheritance - Multilevel Inheritance - Multiple Inheritance - Hierarchical Inheritance - Hybrid Inheritance - Virtual Base Classes - Abstract Classes, Pointers, Virtual Functions and Polymorphism: Pointers - Pointers to Objects -this Pointer - Pointers to Derived Classes - Virtual Functions - Pure Virtual Functions

OPERATOR OVERLOADING:

Operator overloading is one of the exciting features of C++ language. The mechanism of giving special meaning to operators is called as **operator overloading.** It provides a flexible option for the creation of new definitions for most of the C++ operators. All the operators can be overloaded except the following:

1. **Class member access operator (…*)**
2. **Scope resolution operator (: :)**
3. **Size operator (size of)**
4. **Conditional operator (?:)**

Defining Operator Overloading:

Operator overloading is accomplished with the help of a special function, called operator function, which describes the task. Operator overloading can be carried out by means of either member functions or friend functions.

- ➢ A friend function will have only one argument for unary operators and two for binary operators, while member function has no arguments for unary operator and one for binary operators
- ➢ This is because the object is sued to invoke the member function is passed implicitly and there fore it is available for member function. This is not the case with friend functions
- ➢ Arguments may be passed either by value or reference.

The general form is

```
return type class name : : operator op(arg list)
{
      Function body
}
```

Where

ì **return type** is the type of value returned by the specified operation

→ **op** is the operator being overloaded

→ The op is preceded by the keyword operator

→ **Operator op** is the function name

The process of overloading involves the following steps:

1.Create a class that defines the data type that is to be use in the overloading operation.

2.Declare the operator function operator op () in the public part of the class. It may be either a member function or a friend function.

3.Define the operator function to implement the required operations.

OVERLOADING UNARY OPERATORS:

Unary operator overloaded by member function takes no formal arguments where as when they are overloaded by friend functions they take a single argument. A minus operator when used as unary operator takes just one operand.

```
Eg:  #include<iostream.h>
     #include<conio.h>
```

```
       class minus


      {
          int x;
          int y;
          int z;
     public:
     void getdata (int a,int b,int c)
     {
       x = a;
       y = b;
       z = c;
     }
   void putdata( )
   {
   cout<< x<< “\t”<< y<<“\t”<<z<<endl;
   }
   void operator -( )
  {
   x = -x;
   y = -y;
   z = -z;
  }
int main( )
{
minus m1;
clrscr( );
cout<< “ overloading unary minus”<< endl;
m1.getdata (1,2,-3)
cout<<” Before overloading”<< endl;
m1.putdata();
-m1;
cout<<” After overloading”<< endl;
m1.putdata();
return 0;
}
```

OVERLOADING BINARY OPERATORS:

Binary operators are overloaded by means of member functions take one formal argument which is the value to the right of the operator.

```
      #include<iostream.h>
       #include<conio.h>
       class complex
      {
         int x;
          int y;
     public:
     void getcomplex ()
     {
       x = 5;
       y = 5;
```

```
    }
  void putcomplex( )
  {
  cout<< x<< “\t”<< y<<endl;
  }
  complex operator +(operator);
  };
 complex complex : : operator +(complex c)
 {
 complex temp;
 temp.x=x+c.x;
 temp.y=y+c.y;
 return (temp);
 }

 int main( )
{
 complex c1,c2,c3;
clrscr( );
c1.getcomplex( );
c2.getcomplex( );
c3=c2+c1;
c1.putcomplex;
c2.putcomplex;
c3.putcomplex;
}
```

The features of overloading binary operator are

- **It receives only one complex type argument explicitly**
- **It returns a complex type value**
- **It is a member function of complex**

OVERLOADING BINARY OPERATORS USING FRIENDS:

1. A friend function may be used in the place of member functions for overloading binary operator, the only difference being that a friend requires two arguments to be explicitly passed to it, while a member function requires only one.
2. There are certain situations where a friend function is used instead of a member function
3. An object need not be used to invoke a friend function but can be passed as an argument

MANIPUALTION OF STRINGS USING OPERATORS:

1.C++ permits us to create our own definition of operators that can be used to manipulate the strings very much similar in decimal numbers

2.Strings can be defined as class objects, which can be then manipulated like built-in-types. Since the strings vary greatly in size, we use new to allocate memory for each string and a pointer variable to point to the string array

3.We must create string objects that can hold these two pieces of information, namely length and location that are necessary for string manipulations

4.A typical string will look as follows:

```
Class string
{



  char *p;
  int len;
public:
```

```
          ----------
          ----------
     };
```

RULES FOR OVERLOADING OPERATORS:

It is simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1) Only existing operators can be overloaded.New operators cannot be created
2) The overloaded operator must have at least one operand that is of user defined type
3) The basic meaning of operator cannot be changed.It is possible to redefine the plus(+) operator to subtract one value form another
4) Overloaded operators follow the syntax rules of the original operators .They cannot be overridden.
5) There are some operators that cannot be overloaded

• **Size of operator**	**sizeof**
• **Member ship operator**	**.**
• **Pointer to member operator**	**.***
• **Scope resolution operator**	**: :**
• **Conditional operator**	**?:**

6) Friend functions cannot be used to overload certain operators .However, member functions can be used to overload them

• **Assignment operator**	**=**
• **Function Call operator**	**( )**
• **Subscripting operator**	**[ ]**
• **Class member access operator**	**->**

7) Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but those overloaded by means of a friend function, take one reference argument

8)Binary operator overloaded through a member function takes one explicit argument and those, which are overloaded through friend function, take two explicit arguments

9)When using binary operators overloaded through a member function ,the left hand operand must be an object relevant to the class

10)Binary arithmetic operators such as +,-,*,and / must explicitly return a value.They must not attempt to change their own arguments

TYPE CONVERSIONS:

When constants and variables of different types mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of the data to the right of an assignment operator is automatically converted to the type of the variable on the left.

For e.g. the statements

```
Int m;
Float x= 3.141519;



M=x;
```

Convert **x** to an integer before its value is assigned to **m.**

TYPES OF COVERSIONS

There are three types of conversion between incompatible types

1. **Conversion from basic type to class type**
2. **Conversion from class type to basic type**
3. **Conversion from one class type to another class type**

Basic To Class Type:

The conversion from basic type to class type is easy to accomplish .For e.g., a constructor was used to build a vector object from an **int** type array. Consider the following constructor

```
String:: string(char *a)
{
  length= strlen(a);
   p=new char [length +1];
  strcpy(p,a):
}
```

This constructor builds a string type object from a **char*** type variable **a** .The variable length and p are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char* type to string type

Example:

```
 String s1;
Char*name1="IBM PC";
 S1=string (name1);
```

Class To Basic Type:

The constructors are used in type conversion from a basic to class type. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator, usually referred to as a conversion function is :

```
Operator type name()
{
   function statement;
 }
```

This function converts a class type data to typename. The casting operator function should satisfy the following conditions.

- It must be a class member
- It must not specify a return type
- It must not have any arguments

Since it is member function, the object invokes it and therefore, the values used for conversion inside the function belong to the object that invoked the function.This means that the function does not need an argument

One Class To Another Class Type:

Either a constructor or a conversion function can carry out conversions between objects of different classes. The compiler treats the same way.

Example: obj X =Obj Y

Obj X is an object of class **X** and **obj y** is an object of class **y.**

The class Y type data is converted to class X type data and the converted value is assigned to the obj x. Since the conversion takes from class Y to class X,y is known as the source class and X is known as the destination class. We know that the casting operator function

Operator type name ();

Converts the class object of which it is a member to **type name.** The **type name may** be a built-in-type or a user -defined one. In the case of conversions between objects, **typename** refers to the destination class. Therefore, when a class needs to be converted, a casting function can be used. The conversion takes place in the source class and the result is given to the destination class object

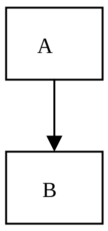
Conversion required	**Conversion takes place in**	
	Source class	**Destination class**
Basic-> class	**Not Applicable**	**Constructor**
Class->Basic	**Casting operator**	**Not Applicable**

Class -> Class	**Casting operator**	**Constructor**

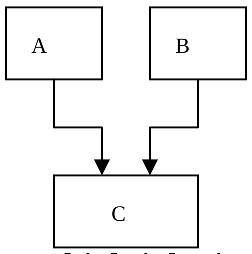
INHERITANCE EXTENDING CLASSES:
Inheritance is an important feature of object-oriented programming. It is a process of creating new classes from existing one. Existing class is known as base class. Newly created class is known as derived class. The mechanism of deriving anew class from an old one is called inheritance. The derived class inherits some so all of the traits from the base class. A class can also inherit properties from one more than one class of from more than one level. A derived class with one base class is called **single inheritance** and one with several base classes is called **multiple inheritance.**

On the other hand, the traits of one class may be inherited by more than one class. This process is known as **Hierarchical inheritance.** The mechanism of deriving classes from another derived class is known as **multilevel inheritance.**

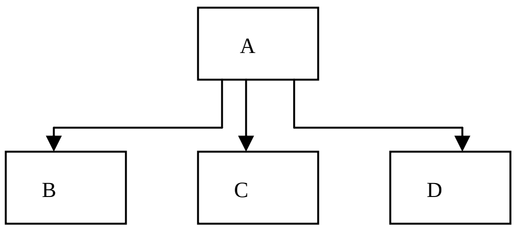
Types Of Inheritance:



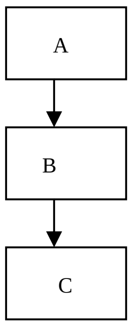
Single Inheritance



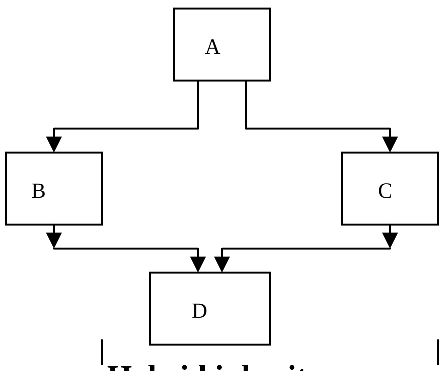
Multiple inheritance



Hierarchical inheritance



Multilevel inheritance



Hybrid inheritance

DEFINING DERIVED CLASSES:
A derived class can be defined by specifying its relationship with the base class in addition to its own details.
A derived class consists of the following components :

- **Keyword class**
- **Name of the derived class**
- **A single colon**
- **Type of derivation(Private,public or protected)**
- **The name of the class**
- **Remainder of the class definition**

The general form of a defining a derived class :

```
Class derived-class-nmee :visibility-mode base-class-name
{
    ------------------------------
    ----------------------------- // member of derived class
};
```

- The colon indicates that the derived class name is derived from the base class name
- The visibility- mode is optional and if present ,it may be either private or public
- The Default visibility mode is private
- Visibility mode specifies whether the features of the base class are privately derived or publicly derived

Example :

```
(1)  Class ABC : private xyz// Private derivation
       {
       members of ABC
       };
(2)   Class ABC : public xyz// public derivation
       {
       members of ABC
       };
(3) Class ABC : xyz// Private derivation  by default
       {



       members of ABC
       };
```

Privately inherited

1. When a **base class** is **privately inherited** by a derived class,' **public'** members of the base class become **'private** members of the derived class and therefore the public members of the base class can only be accessed by the member functions of then derived class.
2. They are inaccessible to the objects of the **derived class.**
3. A public member of a class can be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the **derived class.**

Publicly inherited:

1. On the other hand when the **base class** is **publicly inherited** ,' **public members'** of the base class become '**public members'** of the derived class and therefore they are inaccessible to the objects of the **derived class**
2. In both the cases, the private members are not inherited and therefore, the private members of the **base class** will never become the members of its **derived class**
3. In inheritance, some of the base class data elements and member functions are 'inherited' into the derive class. We can add our own data and member functions and thus extend the functionality of the base class
4. Inheritance, when used to modify and extend the capabilities of the existing classes, become very powerful tool for incremental program development.

SINGLE INHERITANCE:

A derived class with only one base class is called **single inheritance.**

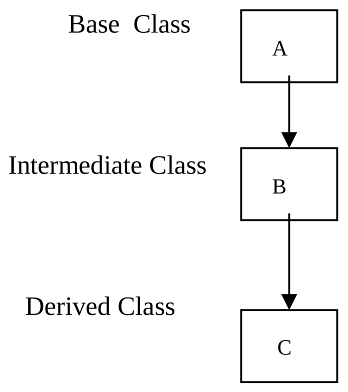
Example :

```
#include<iostream.h>
#include<conio.h>
class student
```

```
{
Int rno;
Char name[ 20];
Int age;
Public:
Void getdata ( )
{
cout<<”Enter the Roll no”;
cin>> rno;
Cout<<”Name”;
Cin>> name;
Cout<<”age”;
Cin>>age;
}
void putdata ( )
{
cout<<”roll no”<< rno;
cout<<”name”<<name;
cout<<”age”<,age;
}
};
class sports :public student
{
int height;
int weight;
public:


void gethtwt ( )
{
cout<<”Enter the height and weight “;
cin>>height>>weight;
}
void puthtwt( )
{
cout<< “Height:”<< height;
cout<<”Weight:”<<weight;
}
};
int main()
{
sports s1;
clrscr ( );
s1.getdata( );
s1.gethwt ( );
s1.putdata( );
s1.puthtwt( );
getch ( );
return 0;
}
```

The class sports is a public derivation of the base class student. Therefore,sports inherits all the public members of student and retains their visibility. Thus a public member of the base class student is also a public member of the derived class sports. The private members of student cannot be inherited by sports. The class sports, in effect will have more members than what it contains at the time of declaration.

MULTILEVEL INHERITANCE:



The process of deriving a class from another derived class is known as multilevel inheritance. The class A serves as a base class for the derived class B,which in turn serves as a base class for the derived class C.The class B is known as intermediate base class since it provides a link for the inheritance between A and C.The chain ABC is known as **Inheritance path.**

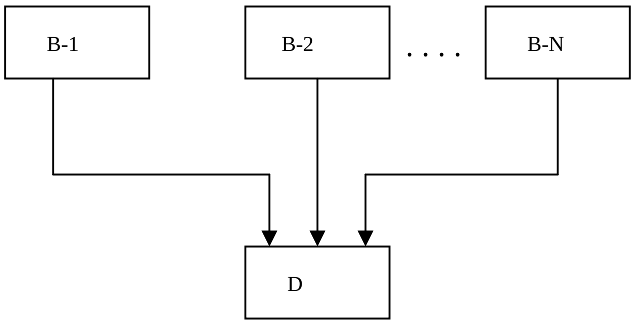
Example for multilevel inheritance:

```
#include<stdio.h>


#include<conio.h>
class student
{
protected:
    int roll-number
public:
    void get-number(int);
    void put number(void);
};
void student : : get-number(int a)
{
  roll-number=a;
}
void student : : put number( )
{
  cout<< "roll number :"<<roll-number;
}
class test : : public student
{
protected:
      float sub1;
      float sub2;
public :
      void get-marks(float, float);
      void put-marks(void);
};
void test : : get-marks(float x, float y)
{
  sub1 = x;
  sub2= y;
}
void test : : put-marks( )
```

```
{
cout<<sub1<<"\n";
cout<<sub2<<"\n";
}
class result : public test
{
 float total;
public:
 void display(void);
};
void result ::display(void)
{
total = sub1 +sub2;
put-number( );
put-marks( );
cout<<total<<"\n";
}
int main ( )


{
result student1;
student1.getnumber(111);
student1.getmarks(75.0,59.5);
student1.display( );
return 0;
}
```

MULTIPLE INHERITANCE:

Multiple inheritance is the process of creating a new class from more than one base classes. It allows us to combine the features of several existing classes as a starting point for defining new class.



The syntax of a derived class with multiple base classes is as follows:

```
Class D : visibility B-1, visibility B-2…..
{
-----------------
-----------------    ( body of D)
-----------------
}
```

Where visibility may be either public or private.The base classes are separated by commas.

Sample program :

```
#include<stdio.h>
#include<conio.h>
class student
{
```

```
protected :
int rno;
char name[20];
public :
void getdata( )
{
cout<< " enter the Rollno and name";
cin>> rno>> name;
}
void putdata ( )
{
coutl<< :"<< rno<," name:"<< name;
}



};
clas marks : public student
{
protected :
int sub1;
int sub2;
public:
void getmarks( )
{
 cout<< "enter the marks";
 cin<< sub1;
 cout<< "enter the marks";
 cin<< sub2;
}
void putmarks()
{
cout<< "English"<<sub1<<"Tamil"<<sub2;
}
};
class result : public marks
{
int total;
public :
void calculate( )
{
total = sub1+sub2;
cout<<"Total"<<total<<endl;
}
};
int main( )
{
result r1;
clrscr ();
r1.getdata ( );
r1.getmarks( );
r1.putdata( );
r1.putmarks( );
r1.calculate( );
getch( );
return 0;
```

}

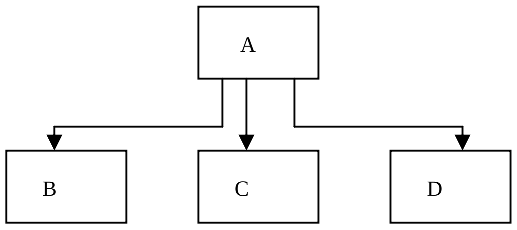
Problems in multiple inheritance:

1.When a function with the same inheritance appears in more than one base class,serious Problems occur
2.Which display( ) function is used by the derived class when we inherit these two classes
3.This problem can be solved by defining named instance within the derived class using the class resolution operator

4.Ambiguity also arise in single inheritance

HIERARCHICAL INHERITANCE:

Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level

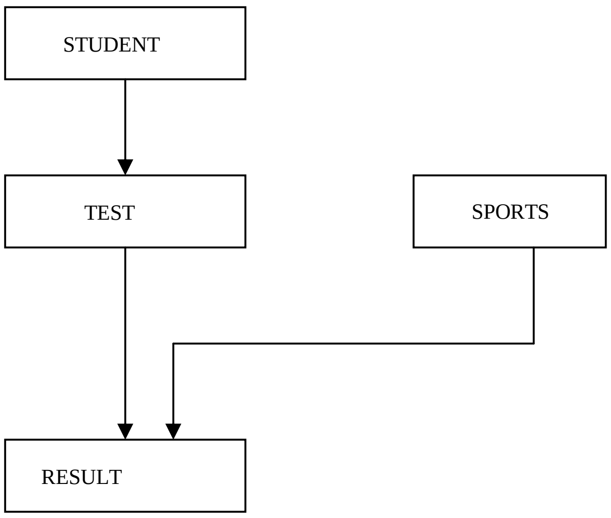


Example :

```
#include<stdio.h>
#include<conio.h>
class student
{
int rno;
char name[20];
int age;
public:
void getdata( )
{
cout<<" Enter the rno,name and age":
cin>>rno>>name>>age;
}
void putdata( )
{
cout<<"\t\t\t"<<rno<<"\t"<<name<<"\t"<<age;
}
};
class sports :public student
{
int height;
int weight;
public:
void getdetails()
{
cout<<"Enter thje height and weight";
cin>>height>>weight;
}
void putdetails()

{
cout<<”\t\t”<<height<<”\t”<<weight;
}
};


clas result :public student


{
char res;
public:
void getres()
{
cout<<”Enter the result of the student”;
cin>>res;
}
void putres( )
{
cout<<res;
}
};
int main()
{
sports s1;
clrscr();
result r1;
s1.getdata();
s1.getdetails();
r1.getres();
s1.putdata();
s1.putdetails();
r1.putres();
getch();
return 0;
}
```

<u>HYBRID INHERITANCE</u>:

There could be situations where we need to apply two or more types of inheritance to design a program. Consider the sample given below. The weight age for sports is stores in a separate class called sports. The result will have both the multilevel and multiple inheritances.

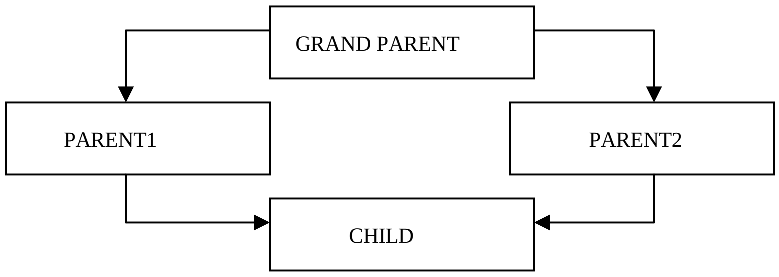


SAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
Class student
{
protected
int rno;
public:
void get-number(int a)
{
rno= a;
}
void put-number(void)
{
cout<<"rollno"<<rno<<endl;
}
};
class test:public student
{
protected :
float part1,part2;
public:
void get-marks(float x,float y)
{
part1=x;
part2= y;
}
void put-marks(void)
{
cout<<"marks obtained:"<<endl;
cout<<"part1"<<part1;
cout<<"part2"<<part2;
}
};


class sports
{
protected:
float score;
public:
void get-score(float s)
{
score=s;
}
void put-score(void)
{
cout<<"sports\t"<<score;


}
};
class result :public test,public sports
{
float total;
public:
void display (void);
};
void result ::display(void)
{
total=part1+part2+score;
put-number()
put-marks();
put-score();
cout<<”Total score”<<total;
}
int main()
{
result student –I;
clrscr();
student-I.get-number(1234);
student-I.get-marks(25,56);
student-I.get-score(6.0)
student-I.display( );
getch();
return 0;
}
```

VIRTUAL BASE CLASSES:

Consider a situation where all the three kinds of inheritance namely **multilevel,multiple and hierarchical inheritances are allowed.**



SAMPLE PROGRAM :

```
 #include<stdio.h>
#include<conio.h>
Class student
{
protected
int rno;
public:
void get-number(int a)
{
rno= a;
}
void put-number(void)
{
cout<<”rollno”<<rno<<endl;
}
};
class test : virtual public student
{
protected :
```

```
float part1,part2;
public:
void get-marks(float x,float y)
{
part1=x;
part2= y;
}
void put-marks(void)
{
cout<<”marks obtained:”<<endl;
cout<<”part1”<<part1;
cout<<”part2”<<part2;
}
};
class sports : publ;ic virtual student
{
protected:
float score;
public:
void get-score(float s)
{
score=s;
}
void put-score(void)
{
cout<<”sports\t”<<score;
}
};
class result :public test,public sports


{
float total;
public:
void display (void);
};
void result ::display(void)
{


total=part1+part2+score;
put-number()


put-marks();
put-score();
cout<<”Total score”<<total;
}
int main()
{
result student –I;
clrscr();
student-I.get-number(1234);
student-I.get-marks(25,56);
student-I.get-score(6.0)
student-I.display( );
```

```
getch();
return 0;
}
```

ABSTRACT CLASSES:

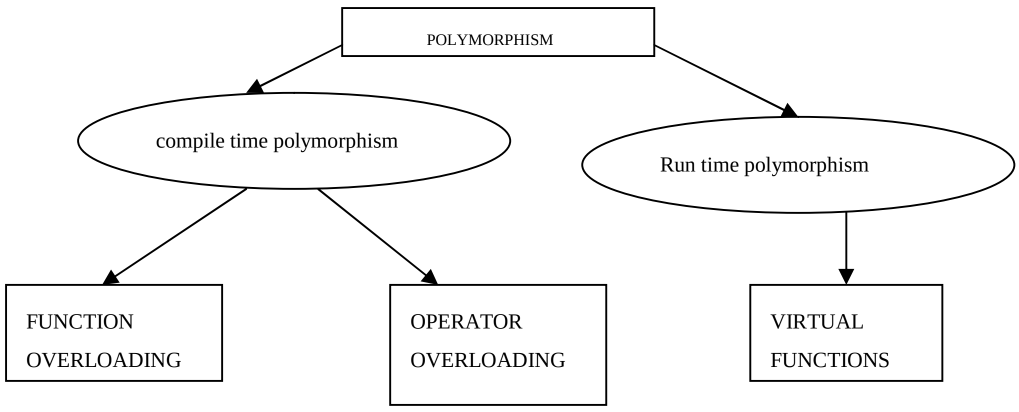
An Abstract class is one that is used to create objects. An abstract class is designed only to act as a base class. It is a design in program development and provides a base upon which other classes may be built.

POINTERS VIRTUAL FUNCTIONS AND POLYMORPHISM:

INTRODUCTION:

Polymorphism:

Polymorphism is one of the crucial features of object oriented programming ***(OOP*).**It simply means ***"one name ,multiple forms".*** The overloaded operators are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called **early binding (Or) static binding or static linking.**Also known as compile time polymorphism ,early binding simply menas that an object is bound to its functiom call at compile time.



POINTERS :

Definition of pointer:

Pointers are one of the key aspects of C++ language. Pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Declaring and initializing pointers:

The declaration of pointer variable takes the following form

Data –type *pointer – variable

Here, a pointer –variable is the name of the pointer, and the data type refers to one of the valid C++ data types such as int, char,float and so on

.The data type is followed by an asterisk(*) symbol, Which distinguishes a pointer variable **from other variables to that compiler. The pointers, which are not initialized in a program are** called **Null pointers**. **Pointers of any data type can be assigned with one value that is '0' called as Null address. Void pointers also known as generic pointers ,which refer to variables of any data type.** Before using void pointers, We must type cast the variables to the specific data types that points to.

Manipulation of Pointers:

A pointer can be manipulated with the **indirection operator,'* '** which is also known as**"Dereference operator".** With this operator, we can indirectly access the data variable content. It takes the following form:

***Pointer-Variable**

Dereferencing a pointer allows to get the content of memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to, change the content of a variable; it can be done with the help of **Dereference operator.**

Example:

```
#include<stdio.h>
#include<conio.h>
Int main()
{
  int a=10,*ptr;
  ptr =&a;
  clrscr();
  cout<< "the value of a"<< a;
  ptr=(ptr)/2;
  cout<<"The value of a<<(*ptr);
  return 0;
}
```

Pointer Expressions and Pointer Arithmetic:

C++ allows pointers to perform the following arithmetic operations:

1.A pointer can be incremented (++) or (--) decremented
2.Any integer can be added or subtracted from a pointer
3.Only pointer can be subtracted from another.

Example:

```
#include<stdio.h>
#include<conio.h>
Int main()
{
int num = [ 10,20,30,40,50];
int*ptr;



int i;
clrscr();
for(i=o; i < 5; i ++)
cout<< num[I];
ptr=num;
cout<<*ptr;
ptr++;
cout<<*ptr;
ptr--;
cout<<*ptr;
cout<<endl;
ptr=ptr+2;
cout<<*ptr;
ptr=ptr-1;
cout<<*ptr;
getch();
return 0;
```

}

Pointers with arrays and strings

Pointers are useful to allocate arrays dynamically.

Difference between array and pointer

Arrays refer to a block of memory space,wheeas pointers do not refer to any section of memory.The memory addresses of arrays cannot be changed,wheeas the contents of the pointer variables,such as the memory addresses that it refer to,can be changed.

Sample program:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int number[50],*ptr;
int n,j;
int sum=0;
clrscr();
cout<< " enter the ccount\n";
cin>>n;
for(i=0;i<=n;i++)
cin>>number[i];
ptr=number;
for(i=1;i<=n;i++)
{
if( *ptr%2 == 0)
sum=sum+*ptr;
ptr++
}
cout<<"\n sum of even number"<<sum<<endl;
getch();
return 0;
}
```

<u>Arrays of pointers</u>

An array of pointer can be created like other variables. It represents a collection of addresses. By declaring array of pointer, we can save a substantial amount of memory space. An array of pointes point to an array of data items. Each element of the pointer array points to item of the data array. An array of pointers can be declared as follows:

int *inarray[10]

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray [1], and the final pointer points to inarray[9].

<u>Pointers and strings</u>

C++ allows us to handle the special kind of arrays. A string is a one dimensional array of characters, which start with the index 0 and ends with the null character'\0' in C++.String operations are performed by using pointers to arrays and then using pointer arithmetic.

<u>Pointers and functions</u>

The concept of pointer to function acts as a base for pointers to members. The pointer to function is known as call back function. These function pointers can be use to refer to a function. Using function pointers, a C++ program can select a function dynamically at run time. A function can also be passed as an argument t o another function. The function pointer cannot be dereferenced.C++ allows comparing two function pointers++ provides two types of unction pointers; function pointers that point to static member functions and function pointers that point to non static members. These two function pointers are incompatible with each other. The function pointers that point to the non-static member function require hidden argument.

```
#include<stdio.h>
#include<conio.h>
class BC
{
public:
   int b;
void show()
{
cout<< " b= "<< b;
}
};
class DC : public  BC
{
public:
   int b;
void show()
{
cout<< " b= "<< b;
cout<<"d= "<<d;
}
};

int main()
{

BC *bptr;
BC base;
bptr= &base;
bptr->b=100;
cout<< bptr->show();
DC derived;
bptr= &derived;
dptr->d=300;
cout<< bptr->show();
DC *dptr;
dptr= &derived;
dptr->d =300;
cout<< dptr->show();
getch();
return 0;
}
```

<u>Virtual functions</u>:

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual .C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

<u>Rules for virtual function</u> :

- The virtual functions must be member's of some class.
- They cannot be static members.
- There are accessed by using object pointers.
- A virtual function can be friend of another class.
- A virtual function in abase class must be defined,even though it may not be used.

- The prototypes of the base class version of avirtual function and all the derived class versions must be identical.
- We cannot have virtual constructors ,but we can have virtual destructors
- While a base pointer can point to any type of derived object ,the reverse is not true.
- When a base pointer points to a derived class,incrementing it will not make it to point to the next object of the derived class.
- If a virtual function is defined in the base class,it need not be necessarily redefined in the derived class

PURE VIRTUAL FUNCTIONS

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task.It only serves as a place holder. such functions are called “do-nothing” functions. A do nothing function may be defined as follows:

Virtual void display( )= 0;

Such functions are called **pure virtual functions**

A pure virtual function is a function declared in abase class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. A class containing pure virtual functions cannot be used to declare any objects of its own .such classes are called abstract classes. The mainobjective of an abstract base class is to provide some traits to the derived classes and to create a base pointer require for achieving run time polymorphism.

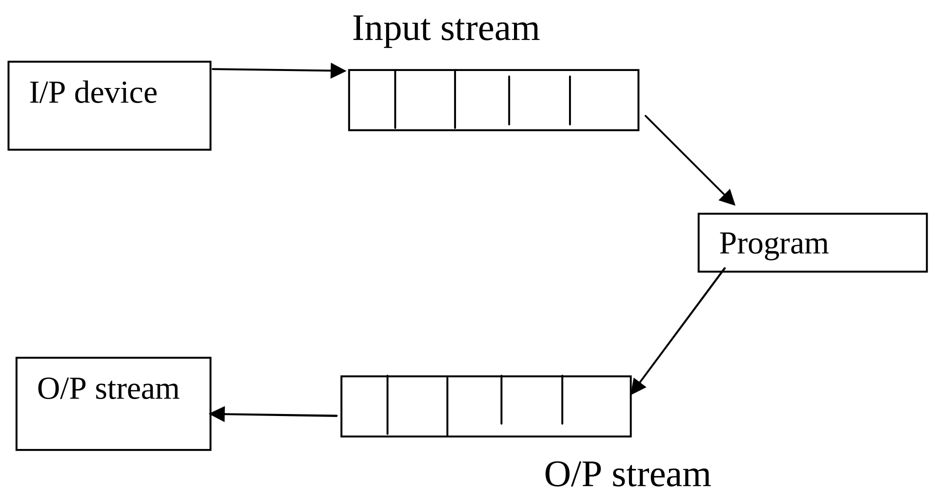
UNIT – IV

Managing I/O Operations: C++ Streams - C++ Stream Classes - Unformatted I/O and Formatted I/O Operations - Managing Output with Manipulators. Working with Files: Classes for File Stream Operations - Opening and Closing a File - Detecting end-of-file - File Pointers and Their Manipulators - Sequential I/O Operations - Updating a File - Error Handling during File Operations - Command Line Arguments

Managing console I/O operation:

C++ Streams:

- The I/O system in C++ is designed to work with a wide variety of devices including terminals,disks,and tape drives. Although each device is very different the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as stream.
- A stream is a sequence of bytes. It acts either as a source from which the input data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called output stream.



C++ STREAM CLASSES:

- The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. The figure shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.
- The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream provides the facilities for formatted O/P.

- The class iostream provides the facilities for handling both input and output streams. Three classes namely, istream_withassign, ostream_withassign, and iostream_withassign add assignment operators to these classes.

<u>Unformatted I/O operation</u>

We have used the object cin and cout for the input and output of various types. This has been made possible by overloading the operators >> and << to regonize all the basic C++ types.

The >> operator is overloaded in the istream class and << is overloaded in the ostream class.

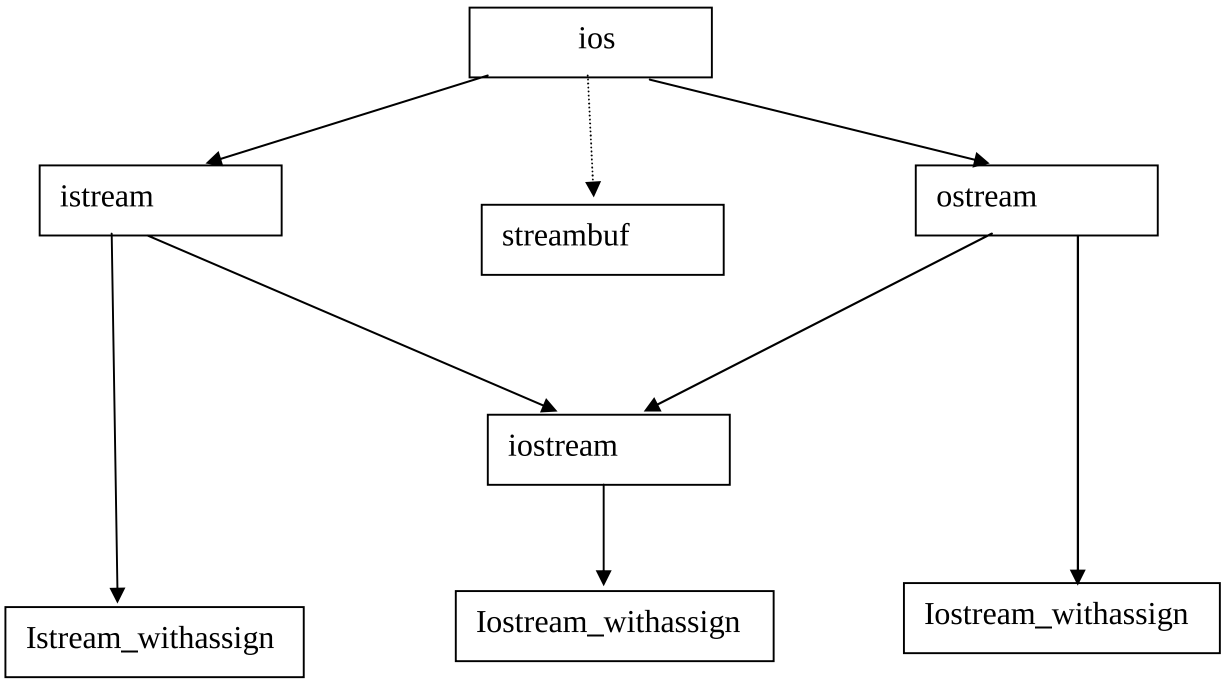
General format for reading data from the keyboard:

```
Cin>>variable1>>variable2>>……>>variableN
```

Variable1, variable2, .. are valid C++ variable names that have been declared already . This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be
Data1, data2,…dataN
The input data are separated by white spaces and should match the type of variable in the cin list spaces, newlines and tabs will be skipped.
Eg
Int code;
Cin>>code;
Suppose the following data is given as input
4258D
The operator will read the character upto 8 and the value 4258 assigned to code. The character D remain in the input stream and will be input to the next in statement . The general form for displaying data on the screen is

```
Cout<<item1<<item2<<…………<<itemN
```

The items item1 through itemN may be variables or constants of any basic type



ig: Stream classes for console I/O operations.

put() and get() functions:

The classes istream and ostream define two member functions get() and put() to handle the single character input/output operations.

There are two types of get() functions:

- ❖ get(char*)
- ❖ get(void)

get(char*)

The get(char*) version assigns the input character to its argument. The get(void) version returns the input character.

```
char c;
cin.get(c);
while(c!='\n')
{
cout<<c;
cin.get(c);
}
```

get(void)

The get(void) version is used as follows:

```
 char c;
cin.get();
```

The value returned by the function get() is assigned to the variable c.

```
#include<iostream.h>
void main()
{
int count =0;char c;
cin.get( );
while (c!='\n')
{
cout.put( );
count++;
cin.get( );
count <<count;
}
```

```
Input:
Object oriented programming
Output
Object oriented programming
Number of characters :27
```

getline() and write() functions:

To read and display a line of text more efficiently using the functions getline() and write().

Getline():

The getline() function reads a whole line of text that ends with a new line character. This function can be invoked by using the object cin as follows:

cin.getline(line,size);

This function call invokes the function get line() which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size1 characters are read. The new line character is read but not saved.

```
#include<iostream>
Int main()
{
        Int size=20;
        Char city[20];
        Cout <<"Enter city name:\n";
        Cin >> city;
        Cout<<"City name:"<<City<<"\n\n";
        Cout<<"Enter another City name:\n";
        Cin.getline(City,size);
        Cout<<"New city name:"<<city<<"\n\n";
        Return 0;
}
```

Output:

```
First run:
Enter city name: Delhi
City name:Delhi
```

```
Second run:
Enter city name: New Delhi
City name: New
Enter city name again:
City name now: Delhi
```

Write()

The write() function displays an entire line and has the following form:

cout.write(line,size);

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display.

```
#include<iostream.h>
#include<string>
Int main()
{
Char * string1=”c++”;
Char * string2=”programming”;
Int m =strlen(string1);
Int m =strlen(string2);
For (int i=1;i<n;i++)
{
Cout.write(string2,i);
Cout <<”\n”;
}
For(i=n;i>0;i--)
{
Cout.write(string2,i);
Cout<<”\n”;
}
Cout.write(string1,m).write(string2,n);
Cout<<”\n”;
Cout.write(string1,10);
Retrun 0;
}
```

Output:

```
P
Pr
Pro
Prog
Progr
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progr
Prog
Pro
Pr
P
```

Formatted Console I/O operations

C++ supports a number of features that could be used for formatting the output. These features include

- ios class functions and flags
- Manipulators
- User-defined O/P functions.

The ios class contains a large number of member functions that help us to format the output in a number of ways. The most important ones among them are listed in the table below.

Function	Task
Width()	To specify the required field size for displaying an o/p value.
Precision()	To specify the number of digits to be displayed after the decimal point of a float value.
Fill()	To specify a character that is used to fill the unused portion of a field.
Setf()	To specify format flags that can control the form of o/p display.
Unsetf()	To clear the flags specified.

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. The table shows important manipulators functions that are frequently used. To access manipulators, the file iomanip should be included in the program.

Manipulators	Equivalent ios fn
Setw()	width()
Setprecision()	precision()
Setfill()	fill()
Setiosflags()	setf()
Resetiosflags()	unsetf()

field width:width()

The width() function is used to define the width of a field necessary for the output of an item. It can be given as:

cout.width(w);

where w is the field width. The output will be printed in a field of w characters wide at the right end of the field.

For eg: **cout.width(5);**
cout<<543<<12;

output:

```
     5    4    3    1    2
```

<u>Setting Precision:precision()</u>

We can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This is done by using the precision() member function as follows:

cout.precision(d);

where d is the number of digits to the right of the decimal point.

For eg:

```
cout.precision(3);
cout<<sqrt(2);
cout<<3.14159;
cout<<2.50032
```

will produce the output as

```
1.141
3.142
2.5
```

<u>Filling and Padding:fill()</u>

We can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

cout.fill(ch);

where ch represents the character which is used for filling the unused positions.

Eg:

```
Cout.fill('*');
Cout.width(10);
Cout<<5250<<"\n";
```

The O/P is:

```
*    *    *    *    *    *    5    2    5    0
```

<u>Formatting Flags, Bit-fields and setf()</u>

The setf(), a member function of the ios class, stands for set flags can be used as:

cout.setf(arg1, arg2)

The arg1 is one of the formatting flags defined in the class ios. The formatting flag specifies the format action required for the output.arg2, known as bit field specifies the group to which the formatting flag belongs

Eg:

cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::float field);
Example:
cout.fill(‘*’);
cout.setf(ios::left,ios::adjustfield);
cout.width(10);
cout<<”table”<<”m”;
this will produce the following output

```
T    A    B    L    E    *    *    *    *    *
```

The statements
Cout.fill(‘*’);
Cout.precision(3);
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::float field);
cout.width(10);
cout<<-12.34567<<”\n”;
will produce the following output

```
-    *    *    *    *    *    2    3    5    E
```

<u>Displaying Trailing zeros and plus sign:</u>

The setf() can be used with the flag ios::showpoint as a single argument to achieve this form of output.For eg:

Cout.setf(ios::showpoint);   //display trailing zeros

would cause cout to display trailing zeros and trailing decimal point.
Similarly, a plus sign can be printed before a positive number using the following statement:

Cout.setf(ios::showpos);   // show + sign

For eg, the statements

Cout.setf(ios::showpoint);
Cout.setf(ios::showpos);
Cout.precision(3);
Cout.setf(ios::fixed, ios::floatfield);
Cout.setf(ios::internal, ios::adjustfield);
Cout.width(10);
Cout<<275.5<<”\n”;

The flags such as showpoint and showpos do not have any bit fields and therefore are used as single arguments in setf() . This possible because the setf() has been declared as an overloaded function in the class ios

Note

- the flags set by setf() remain effective until they are reset or unset
- a format flag can be reset any number of times in a program
- we can apply more than one format controls jointly on an output value,
- The setf() sets the specified flags leaves other unchanged.

Managing output with manipulators

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output formats. They provide the same features as that of the ios member functions and flags. Two or more manipulators can be used as a chain in one statement as shown below:

Cout<<manip1<<manip2<<manip3<<item;
Cout<<manip1<<item1<<manip3<<item2;

To access these manipulators we must include the file iomanip.h in the program. The most commonly used manipulators are

Manipulator	**Meaning**
Setw(int w)	Set the field width to w.
Setiosflags(long f)	Set the format flag f.
Setfill(int c)	set the fill character to c.

Example:

cout<<setw(10)<<12345;

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows

Cout<<setw(10)<<setioflags(ios::left)<<12345;

One statement can be used to format output for two or more values. For example, the statement

Cout<<setw(5)<<set precision(2)<<1.2345;
Cout<<setw(10)<<set precision(4)<<sqrt(2);
Cout<<setw(15)<<setioflags(ios::scientific)<<sqrt(2);

Will produce all the three values in one line with the field sizes of 5, 10 and 15 respectively

The formatting of the output values using both manipulators and ios functions program are as follows

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<iomanip.h>
void main()
{
clrscr();
cout.setf(ios::showpoint);
cout<<setw(5)<<"n"<<setw(15)
      <<"Inverse of n"<<setw(15)
       <<"sum _of_terms\n\n";
Double term,sum=0;
For(int n=1;n<10;n++)
{
Term=1.0/float(n);
Sum=sum+term;
Cout<<setw(5)<<n<<setw(14)
    << Setprecision(4)
   <<setiosflags(ios::scientific)<<term

<<setw(13)<<resetiosflags(ios::scientific)
  <<sum<<end1;
}
Return 0;
}
```

Output:

```
N  Inverse_of_n    Sum_of_terms
1   1.0000e+000    1.0000
2   5.0000e+001    1.5000
```

<u>Designing our own Manipulators:</u>

We can design our own manipulators fro certain special purpose.

The general form for creating a manipulator without any arguments is:

```
ostream & manipulator(ostream &output)
{       ……..
        …….
        …….(code)
        return output;
}
```

Here, the manipulator is the name of the manipulator under creation. The following function defines a manipulator called unit that displays "inches".

```
ostream &unit(ostream&output)
{       output<<"inches";
        return output;
}
```

The statement

```
        cout<<36<<unit;
```

will produce the following output

```
        36 inches
```

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
ostream &college(ostream & output)

{output <<"Rs";
return output;
}

ostream & form (ostream & output)
{
Output.setf(ios::showpos);
Output.setf(ios::showint);
    output.fill ('*');
    output.precision(2);
   output<<setiosflags(ios::fixed)
            <<setw(10);
    return output;
}
```

```
void main()
{   clrscr();
     cout<<currency<<form<< 7864.5;
   return0;
}
```

Output:

```
Rs**+7864.50
```

<u>WORKING WITH FILES:</u>

We need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk.

- Programs can be designed to perform the read and write operations on these files
- Program typically involves either or both of the following kinds of data communication.
- Data transfer between the console unit and the program.
- Data transfer between the program and a disk file.

Console-program file interaction.

- The I/O system of C++ handles file operation which are very much similar to the console input and output operations. It uses file stream as an interface between the program and the files. The stream that support data to the program is known as input stream and the one that receives data from the program is known as output stream.
- In other words, the input stream extracts data from the file and the output stream inserts data to the file. The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly,

the output operations involve establishing an output stream with the necessary links with the program and the output file.

Classes for file stream operations:

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream,ofstream & fstream.

These classes are derived from fstreambase and from the corresponding iostream.h as shown in the figure.

These classes designed to manage the disk file, are declared in fstream.h and therefore we must include this file in any program that uses files.

Opening and Closing of files:

- Suitable name for the file
- Data type and structure
- Purpose
- Opening method

The file name is a string of character that make up a valid file name for the os. It may contain two
parts

primary name and an optional period with extension

eg:

input.data
test.doc

A file can be opened in two ways:

- Using the constructor function of the class.
- Using the member function open() of the class

Opening files using Constructors:

In order to access a file, it has to be opened either in read, write, or append mode.

This involves the following steps:

- Create a file stream object to manage the stream using the appropriate class. That is, the class

ofstream is used to create the output stream and the ifstream to create the input stream.

- Initialize the file object with the desired filename.

Example:

```
ofstream outfile(“results”);
ifstream infile(“data”);
```

Program:

```
#include<fstream.h>
main()
{
   ofstream outf(“ITEM”);
```

```
            cout<<”Enter name:”;
            char name[30];
            cin>>name;
            outf<<name<<”\n”;
            cout<<”Enter item cost:”
            float cost;
            cin >>cost;
            outf.close();
            ifstream inf(“ITEM”);
            inf>>name;
            inf>>cost;
            cout<<”\n”;
            cout<<”Item name:”<<name<<”\n”;
            cout<<”item cost:”<<cost<<”\n”;
            inf.close();
          return 0;
        }
```

Output:

```
Enter item name: CD-ROM
Enter item cost:250
Item name: CD-ROM
Item cost:250
```

opening files using open()

The function open() can be used to open multiple files that use the stream object.

Syntax:

```
        file-stream-class stream object;
        stream-object.open(“filename”);
```

Example:

```
        ifstream infile;
        infile.open(“sam.data”); //opening a file
```

Opening files in Write mode:

```
ofstream outfile;    // create stream for output
outfile.open(“Student.out”);
…….
outfile.close();// disconnect stream from student.out
```

Opening files in Read mode:

```
ifstream infile;       // create stream for input
infile.open(“Student.out”);
…….
```

infile.close();// disconnect stream from student.out

<u>Detecting end of file:</u>

Detection of the end of file condition is necessary for preventing any further attempt to read data from the file.

While (fin)

An **ifstream** object such as **fin** returns a value of 0 if any error occurs in the file operation including the end of file condition.

if (fin1.eof ( ) !=0) { exit(1); }

eof( ) is a member function of **ios** class. It returns a non zero value if the end of file (EOF) condition is encountered and a zero otherwise.

<u>More about open():File Modes</u>

stream-object.open("file-name",mode);

The second argument mode specifies the purpose for which the file is opened.

File mode parameters:

Parameter	**Meaning**
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::out	Open file for writing only
ios::trunc	Delete contents of the file if it exists
ios::nocreate	Open fails if the file does not exists
ios::noreplace	Open fails if the file already exists

Example:

fout.open("data", ios::in);

the mode can combine two or more parameters using the bitwise OR operator.

<u>File pointers and their Manipulations</u>:

The file management system associates two pointers with each file called file pointers. In C++, they are called get pointer (Input pointer) and put pointer (Output pointer).

- ✓ The get pointer specifies a location from where the current reading operation is initiated.

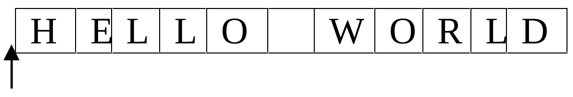
- ✓ The put pointer specifies a location from where the current writing operation is initiated. On completion of a read or write operation, the appropriate pointer will be advanced automatically.
- ✓

<u>Default Actions:</u>

When a file is opened, the logical location of file pointers is shown below:

(1) When a file is opened in read-only mode, the input pointer is automatically set at the beginning so that the file can be read form the start.

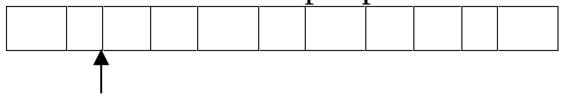
“hello” files



Open for reading only

Input pointer

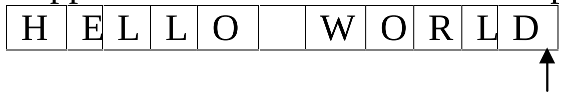
(2) Similarly, when a file is opened in write-only mode, the existing contents are deleted and the output pointer is set at the beginning.



Open for writing only

Output pointer

(3) If an existing file is to be opened to add more data, the file is opened in ‘append’ mode. This moves the output pointer to the end of the file.



Open in append mode(for writing more data)

Output pointer

<u>Functions for manipulation of File Pointers</u>

The C++ I/O system supports four functions for setting a file pointer to any desired location inside the file or to get the current file pointer.

Function	**Action Performed**
seekg	**Moves get pointer to a specified location**
Seekp	**Moves put pointer to a specified location.**
tellg()	**Moves the current position of the get pointer**
tellp()	**Moves the current position of the put pointer.**

<u>Specifying the offset:</u>

- The argument to ‘seek’functions represent the absolute position of the file. ‘seek’ functions seekg() and seekp() can also be used with two arguments as follows:

```
seekg(offset, refposition);
seekp(offset, refposition);
```

- the parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of following three constants defined in the ios class:

  ios::beg---- start of the file.
  ios::cur---- current position of the pointer .
  ios::end---- end of the file.

<u>Seek Calls and their actions:</u>

Seek Call	**Action Performed**
fout.seekg(0,ios::beg)	go to the beginning of the file.
fout.seekg(0,ios::cur)	Stay at the current file
fout.seekg(0,ios::end)	Go to the end of the file
fout.seekg(n,ios::beg)	Move to (n+1) bytes from the current position
fout.seekg(n,ios::cur)	Move forward by n bytes from the current position
fout.seekg(-n,ios::cur)	Move backward by n bytes from the current position
fout.seekg(-n,ios::end)	Move forward by n bytes from the end
fout.seekp(n,ios::beg)	Move write pointer to (n+1) byte location
fout.seekp(-n,ios::cur)	Move write pointer backward by n bytes

Sequential input and ouput operations.

The operations that are used are as follows

- put( )
- get ( )
- write ()
- read( )

<u>Put() and get() functions:</u>

- The function **put()** writes a single character to the associated stream. Similarly, the function **get()** reads a single character from the associated stream. Ex program given “read& writing a string”.

<u>Write() and read() functions:</u>

- The functions write() and read(), unlike the function put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. An int takes two bytes to store its value in the binary form, irrespective of its size. But a 4-digit int will take 4 bytes to store it in the character form

  - The binary format is more accurate for storing the numbers, as they are stored in the exact internal representation. There are no conversions while saving the data therefore saving is much faster.

The binary input and output functions takes the following form:

infile.read ((char *) &v, sizeof(v));
Outfile.write (char *) &v, sizeof(v));

These functions take two arguments. The first is the address of the variable v, and the second is the length of that variable in bytes

Updating a file:

Updating is a routine task in the maintenance of any data file. The updating would include one or more of the following tasks:

- ✓ Displaying the contents of a file
- ✓ Modifying an existing item
- ✓ Adding a new item
- ✓ Deleting an existing item

These actions require the file pointers to move to a particular location that corresponds to the item/ object under consideration. This can be easily implemented if the file contains a collection of items / objects of equal lengths

Error handling during file operations:

The stream state member functions give the information status like end of files has been reached or file open failure and so on. The following stream state member function are used in C++:

- eof()
- fail()
- bad()
- good()

<u>eof():</u>

It is used to check whether a file pointer has reached the end of a file character or not. If it is successful, eof() member function returns a nonzero, otherwise returns 1.

<u>fail():</u>

It is used to check whether a file has been opened for input or output successfully, or any invalid operations are attempted or there is an unrecoverable error.

<u>bad():</u>

It is used to check whether any invalid file operations has been attempted or there is an unrecoverable error. It returns a nonzero if it is true; otherwise returns a zero.

good():

It is used to check whether the previous file operation has been successful or not.

COMMAND LINE ARGUMENT:

It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process.

Eg: C>exam data results

Consider the first line of the main function,

i.e., main ()

These parentheses may contain special arguments that allow parameters to be passed to main from the Operating System. Two such arguments are **argc** and **argv**, respectively.

The first of these, **argc**, must be an integer variable, while the second, **argv**, is an array of pointer to characters, i.e., an array of strings. Each string in this array will represent a parameter that is passed to main. The value of **argc** will indicate the number of parameters passed. The value of argc would be 3 and the argv would be an array of three pointers to strings as shown below.

Arg[0]->exam
Arg[1]->data
Arg[3]->results

Question bank
Unit IV

2 Marks:

1. Explain about C ++ stream classes.
2. Explain about C ++ stream classes.
3. Defining field width:width()
4. Designing our own Manipulators:
5. Explain in detail about random access file.
6. Designing our own Manipulators

4 Marks:

1. Explain in detail about put() and get() function.
2. Explain in detail about getline() and write() functions
3. Write short notes on classes for file stream operations.
4. Explain in detail about opening and closing file.
5. Write short notes on command line argument

12 Marks:

1. Explain in detail about Formatted Console I/O operations with example.
2. Explain in detail about managing output with manipulators
3. Discuss briefly   about file pointers and their manipulations
4. Explain in detail about sequential input and output operations?
5. Write a short note on error handling during file operations

UNIT- V

Templates: Class Templates - Class Templates with Multiple Parameters - Function Templates - Function Templates with Multiple Parameters - Overloading of Template Functions - Member Function Templates-Non-Type Template Arguments. Exception Handling: Basics - Exception Handling Mechanism - Throwing Mechanism - Catching Mechanism - Rethrowing an Exception - Specifying Exceptions

TEMPLATES:

- Templates is one of the feature added to C++.
- C++ supports a mechanism known as template to implement the concept of generic programming.
- Template used to define generic and functions, and thus provides supports for generic programming.
- Generic Programming is an approach where generic types are used as parameters in algorithms.
- A template can be considered as a kind of macro.
- When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.
- A template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes are called parameterized classes or functions.

CLASS TEMPLATES

- Templates allows to define generic classes. It is a simple process to create a generic class using a template with an anonymous type.
- The general format of a class template is
  template <class T>
  class classname
  {
  //class member specification
  //with anonymous type T
  //wherever appropriate
  //----------------
  };
- The class template definition is very similar to a ordinary class definition except the prefix template
- **<class T>** and the use of type **T.**
- This prefix tells the complier that we are going to declare a template and use T as a type name in the declaration.
- A class created from a class template is called a ***template class.***
- The syntax for defining an object of a template class is: class name <type> object name (arglist); ***classname <type>objectname (arglist);2m***
- This process of creating a specific class from a class template is called ***instantiation.***
- The complier will perform the error analysis only when an instantiation takes place.

- It is therefore, advisable to create and debug an ordinary class before converting it into a template.

```
#include<iostream.h>
#include<conio.h>
Template <class>
class addition
{
        T a:
        T b;
public:
void getdate ()
{
Cout<<”Enter the First Number”<<endl;
cin>>a;
cout<<”Enter the Second Number”<<endl;
cin>>b;
}
Void adddata()
{
T tl;
T1=a+b;
count<<”The Result is ”<<tl<<endl;
}
Void putdata()
{
Count<<”First Number: “<<a<<”\n”;
        cout <<”Secound Number:”<<b<<”\n”
        }
};
int main ()
{
Addition <int>o1;
Addition<float>o2;
Clscr();
o1.getdata();
o1.putdata();
o1.adddata();
o2.getdata();
o2.getdata();
o2putdata();
o2.adddata();
return 0;
}
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can us more than one generic data type in a class template.
They are declared as a comma-separated list within the template specification as shown below:

```
template <class T1,classT2,……..>
        class classname
        {
        ………….
        ………….  (Body of the class)
        ………….
        };
```

Example:

```
        #include <iostream.h>
        #include <class T1,classT2>
        class addition
                T1 a;
                T2 b;
        Public:
        void getdata()
        {
        cout<<"Enter the First Number"<<endl;
         cin>>a;
        cout<<"Enter the Second Number"<<endl;
        cin>>;
        }
        void adddata()
        {
        T1 t1;
        T1=a+b;
        cout<<"The Result is "<<t1<<endl;
        }
Void pudata()
        {
        cout<<"First Number: "<<a<<"\n"
        cout<<"Second Number:"<<b<<"\n":
        }
        };
        int main()
        {
        Addition <int,float>o1;
        addition <float,double>o2;
        clrscr();
        o1.getdata();
        o1.putdata();
        o1.adddata();
        o2.getdata();
        o2.putdata();
        o2.adddata();
```

```
retun 0;
}
```

<u>FUNCTION TEMPLATES</u>:

- A function template can be created that could be used to create a family of functions with different argument types.
- The general format of a function template is:
  ```
  template <classT>
  return type functionname (argument of type T)
  {
  \\ ………….

  \\Body function
  \\ with type T
  Wherever appropriate
  \\………….
  }
  ```
- The function template syntax is similar to that of the class template except that we are defining function instead classes.
- We must use the template parameter T as and when necessary in the function body and in its argument list.

Example Program:

```
        # include <iostream.h>
        # include <conio.h>
        template <class T>
        {
        void swap(T & b1)
        {
        T temp;
        Temp=a1;
        a1=b1;
        b1=temp;
        }
        int temp;
        {
        int a,b;
        clrscr();
cout <<"Before Swapping"<<end;
        a=12;b=23;
        cout<<a<<"\t"<<b<<endl;
        swap(a,b);
        cout<<a<<"After Swapping"<<endl;
        cout<<a<<"\t"<<b<<endl;
        floatc,d;
        c=12.3;13.4;
        cout<<"Before Swapping"<<endl;
        cout<<c<<"\t"<<endl;
        swap(c,d);
        cout<<"After Swapping"<<endl;
```

```
cout<<c<<"\t"<<endl;
return 0;
}
```

FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS

Like template classes, more than one generic data type can be in used template statement using a comma-separated list as shown below:

```
template <classT1,class T2……>
        returntype functions (arguments of types T1, T2,……)
        {
                ………………
                ……………… (Body of function)
                ………………
        }
```

OVERLOADING OF TEMPLATE FUNCTIONS

- A template function may be overloaded either by a template functions or ordinary functions of its name.
- In such cases, the overloading resolution is accomplished as follows:
  - Call an ordinary function that has an exact match.
  - Call template function that could be created with an exact match.
  - Try normal overloading resolution to ordinary functions and call the one that matches.
- An error is generated if no match is found. No automatic type conversions are applied to arguments on the template functions.

```
# include <iostream.h>
# include <conio.h>
Tmplate <class T>
void display (Tx)
{
cout <<"Template Display"<<x<<endl;
}
Void display (inx1)
}
int main ()
{
Clrscr();
display (100);
display('a');
return 0;
}
```

<u>MEMBER FUNCTION TEMPLATES</u>:

- When a class template is created for vector, all the member functions were defined as line, which was not necessary.
- We could have defined them outside the class as well.
- But remember that the member functions of the template classes themselves are parameterized by the type argument (to their template classes) and there other these functions must be defined by the functions template.

It takes the following general form:

```
        Template<classT>
        return-type classname <T>::functionname
        (arglist)
        {
        //…………….
        // Functions body
        //…………….
        }
# include <iostream.h>
# include <conio.h>
template <class T>
class addition
{
        T a;
        T b;
public;
void getdata();
void adddata();
void putdata();
};
Template<class T>
Void addition <T>::getdata()
{
Cout<<"Enter the First Number"<<endl;
cin >>b;
{
template <class T>
void addition <T>::adddata()
T t1;
t1 = a+b;
cout<<"The Result is"<<t1<<endl;
}
    template <class T>
        return-type classname <T>::functionname (arglist)
{
                //…………….
                // Functions body
                //…………….
}
 Template<class T>
void addition <T>::putdata()
```

```
{
cout<<"First Number: "<<a<<"\n";
{
cout <<"First Number:"<<a<<"\n";
cout <<"Second Number:"<<b<<"\n";
}


int main()
        {
        addition <int>o1;
        addition <float>o2;
        clrscr();
        o1.getdata();
        o1.putdata();
        o1.adddata();
        o2.getdata();
        o2.putdata();
        o2.adddata();
        return 0;
        }
```

NON-TYPE TEMPLATE ARGUMENTS

- A template can have multiple arguments.
- It is also possible to use not-type arguments
- In addition to type T, we can also use other arguments such as strings, function names, constant expressions and built in types. Consider the following example

  ```
  Template<class T, int size>
  class array
  {
          T a[size];// automatic array initialization
          //…………….
          //…………….
  };
  ```
- This template supplies the size of the array as an argument.
- This implies that the size array is known to the complier time itself.
- The argument must be specified whenever a template class is created.

Example:

array <int,10> al;
array <flat,5> a2;
array <char,20> a3;

EXCEPTION HANDLING

INTRODUCTION

- The two most common type of bugs are ***logic errors and syntactic errors.***
- The logic errors occur due to poor understanding of the problem and solution procedure.
- The syntactic error arise due to poor understanding of the language itself.

- These errors can be detected by using exhaustive debugging and testing procedures.
- There are some peculiar problems other than logic or syntax errors. They are known as exceptions.
- ***Exceptions*** are runtime anomalies or unusual condition that a program may encounter while executing.
- Anomalies might include conditions such as division b y zero, access to an array outside of its bounds, or running out of memory or disk space.
- When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively.
- Exception handling was not part of the original C++. It is a new feature added.

BASICS OF EXCEPTION HANDLING:

- Exceptions are two kinds, namely ***synchronous exception and asynchronous exception.*** Errors such as "out-of-range index" and "overflow" belong to the synchronous type exceptions.
- The error that are caused by events the control of the program (such as keyboard and interrupts) are called synchronous exceptions.
- The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.
- The mechanism suggests a separate error handling code that performs the following tasks:
  1. ***Find the problem (Hit the exception)***
  2. ***Inform that an error has occurred (Throw the exception)***
  3. ***Receive the error information (Catch the exception)***
  4. ***take corrective actions (Handle the exception)***
- The error handling code basically consists of two segments, one to d\detect errors and to throw exception, and the other to catch the exceptions and to take appropriate actions.

EXCEPTION HANDLING MECHANISM:

- C++, exception handling mechanism is basically built upon three keywords, namely, **try, throw and catch.**
- The key word ***try*** is used to preface a block of statements (surrounded by braces) which may generate exceptions.
- This block of statement is known as try block.
- When an exception is detected, it is thrown using a ***throw*** statement in the try block.
- A ***catch*** block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately.

- The relationship is shown below:

try
Detects and throws an exception

Catch block
Catches and handles the exception

- The catch block that catches an exception must immediately follow the try block that throws the exception.

- When the try block throws an exception, the program control leaves the try block and enters the catch are statement of the catch block.
- Exceptions are object used to transmit information about a problem.
- If the type of object thrown matches the arg type in the catch statement, then catch block is executed for handling the exception.
- If they do not match, the program is aborted with the help of the abort () function which is invoked by default.
- When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped.

Function invoked by try throwing exception:

Throw
Function that causes an exception

Try block
Invokes a function that contains an exception

Catch Block
Catches and handles the exception

```
Type function (arg list) // Function with exception
{       …………….
        throw (object); // Throws exception
        …………….
}
try
{
…………….
……………. // Invoke function here
}
catch (type arg)
{
…………….// Handles exception here
}
```

<u>THROWING MECHANISM</u>:

- When an exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following form:
  throw (exception);
  throw exception;
  throw;
- The operand object exception may be of any type, including constants. It is also possible to throw objects not intended for error handling.
- When an exception is thrown, the catch statement associated with the try block will catch it. That is, the control exists the current try block and is transferred to the catch block after that try block.
- Throw point can be in a deeply nested scope with in a try block or in a deeply nested function call. In any case control is transferred to the catch statement.

<u>CATCHING MECHANISM:</u>

A **catch** block look like a function definition and is of the form catch (type arg)

```
{
        //statement for managing exceptions
}
```

The catch Block that catches an Exception Must Immediately follow the try Block That & Throw the exception

The *type* indicates the type of the exception that catch block handles.

The parameter ***arg*** is an optional parameter name.

Multiple Catch Statement:

- It is possible that a program segment has more than one condition to throw an exception.
- In such cases, we can associate more than one catch statement with a try as shown below:

  ```
  try
  {
          // try block
  }
  catch (type 1 arg)
  {
          //catch block
  }
          catch(type 2arg)
  {
          //catch block
  }
          //catch (type N arg)
  {
          //catch block N
  }
  ```
- When an exception is thrown, the exceptional handles are searched in order for an appropriate mach. The first handler that yielder a match is executed.

- After executing the handle, the control goes to the first statement after the last catch block for that try.
- It is possible that arguments of several catch statements match the type of an exception.
- In such cases, the first handler that matches the execution type is executed.

Sample Program:

```
# include <iostream.h>
# include <conio.h>
void test (int x)
{
try
{
        If (x==1) throw x;
```

```
else
        if (x==0) throw 'x';
else
        if (x==_1) throw1.0;
cout <<"End of try – block\n";
}
cout (char c)
{
cout <<"Caught a character\n";
}
Catch (int m)
{
cout <<"Caught an Integer"\n";
}
catch (double d)
cout <<"Caught a double"<<endl;
}
int main ()
{
cout <<"Multiple Catch"<<endl;
cout <<"x==1"endl;
test (1);
cout <<"x==0"<<endl;
test (0);
cout <<"x==-1"<<endl;
test (-1);
cout<<"x==2"<<endl;
test(2);
cout<<"x==2"<<endl;
test (2);
retrun 0;
}
```

Catch All Exception:

- It is not possible to anticipate all possible types of exceptions and therefore may be not able to design independent catch handlers to catch them.
- In such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement using ellipses as follows:

```
        catch (….)

        {                   //Statement for Processing
        }                   //all exceptions
```

Example Program:

```
        # include <iostream.h>
        # include <conio.h>
        void test (int x)
        {
        try
        {
```

```
if (x==0) throw x;
if (x==-1) throw ‘x’;
if (x==1) throw 1.0;
}
catch (…)
cout <<”Caught  an Exception”;
}
}
int main ()
{
cout <<”Testing Generic Catch”<<endl;
test (-1);
test (0);
test (1);
return 0;
}
```

<u>RETHROWING AN EXCEPTION:</u>

- An handler may decide to throw the exception caught without processing it. In such situations, we simply invoke throw without any arguments as shown below:
    - ***throw***
- This causes the current exception to be throw to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.
- When an exception is rethrown, it not be caught by the same catch statement or any other catch in that group. Rather, it will caught by an appropriate catch in the outer try / catch sequence only.
- A catch handler itself may detect and throw an exception.
- The exception thrown will not be caught by any catch statements in that group. It will be passed on to the next try/catch sequence for processing.

Sample Program:

```
# include <iostream.h>
# include <conio.h>
void divide <double x, double x>
{
        cout <<”inside Function”endl;
        try
        {
                if (y==0.0)
                throw y;
                else


                cout<<””Division=”<</y<<endl;
                }
                catch (double)
                cout <<”Caught double inside function “”endl;
                throw
                }
                cout <<”Inside of function”<<endl;
```

```
}
        int main ()
        {
        Cout<<"Inside Main"<<endl;
        try
{
        divide (10.5,2.0);
        divide (20.,0.0);
}
catch (double)
{
        cout <<"Caught double inside main"<<endl;
}
return 0;
}
cout
cout <<"Caught double inside main"<<endl;
                }
return 0;
}
```

SPECIFYING EXCEPTIONS:

Explain the syntax of specifying Exceptions in C++.
Specifying Exceptions:

- It is possible to restrict a function to throw only certain specified exceptions.
- This is achieved by adding a throw list clause to the function definition.
- The general form of using an exception specification is:

  ```
  type function (arg-list) throw (type-list)
  {
          //Function Body
  }
  ```
- The type-list specifies the type of exception that may be thrown. Throwing any other type of exception will cause abnormal program termination.
- To prevent a function from throwing any exception, the type-list can be made empty./ that is, use
  throw (); //empty list → in the function header line.
- **A function can only be restricted in what types of exceptions is throws back to the try block that called it.**
- **The restriction applies only when throwing an exception out of the function (and not with in a function).**
- A function can only be Restricted in what types of Exceptions is throws Back to the Try Block That called n.
- The Restriction Applies only when throwing an exception out of the function.
  (And not with in a function)

I. The type-list specifies the type of exception that may be thrown. Throwing any other type of exception will cause abnormal program termination.

II. To prevent a function from throwing any exception, the type – list can be made empty. That is, use throw (); // Empty (ist) in the function Header line.