CS - 14 : C++ and Object Oriented Programming

Objectives:

- To provide of OOPs concepts, input/output data management, arrays in C++, functions, classes, objects, pointers, and much more.
- Object-Oriented features, which allow the programmer to create objects within the code.

Prerequisites: concepts of OOPs and their implementation

NO.	Topics	Details	Marks Weight In%	Min lec	Page No.
1	**Principles of object oriented programming Tokens, expressions and control statements**	• Procedure – oriented programming • Object oriented programming paradigm • Basic concepts of object oriented Programming • Benefits of object oriented programming • Application of object oriented programming • What is c++? • Application of c++ • Input/output operators • Structure of c++ program • Introduction of namespace • Tokens : keywords, identifiers, basic data types, user- defined types, derived data types, symbolic constants, type compatibility, declaration of variables, dynamic initialization of variables, reference variables • Operators in C++: scope resolution operator, member referencing operator, memory management operator, manipulators, type cast operator. • Expression : Expression and their types, special assignment operator, implicit conversions, operator precedence	20	15	4
	Functions in C++	• The main function • Function prototype • Call by reference • Return by reference • Inline function • Default arguments • Const arguments • Functions overloading • Adding C Functions turbo C++			
2	**Classes and Objects, Constructor and Destructor**	• C structures revisited • Specifying a class • Local Classes • Nested Classes • Defining member functions, nesting of Member functions, private member function, making outside function inline • Arrays within a class • Memory allocation for objects • Static data member • Static member functions • Arrays of objects	20	12	22

		• Objects as function arguments • Friendly functions • Returning objects • Const member function • Pointer to members • Characteristics of constructor • Explicit constructor • Parameterized constructor • Multiple constructor in a class • Constructor with default argument • Copy constructor • Dynamic initialization of objects • Constructing two dimensional array • Dynamic constructor • MIL , Advantage of MIL • Destructors			
3	**Operator overloading and type conversion, Inheritance**	• Concept of operator overloading • Over loading unary and binary operators • Overloading of operators using friend Function • Manipulation of string using operators • Rules for operator overloading • Type conversions. • Comparison of different method of conversion • Defining derived classes • Types of inheritance (Single, Multiple, Multi-level, Hierarchical, Hybrid) • Virtual base class & Abstract class • Constructors in derived class • Application of Constructor and Destructor in inheritance • Containership, Inheritance V/s Containership	20	11	42
4	**Pointer, Virtual functions and Polymorphism, RTTI Console I/O operations**	• Pointer to Object • Pointer to derived class • this pointer • Rules for virtual function • Virtual function and pure virtual function. • Default argument to virtual function • Run Time Type Identification • C++ streams • C++ stream classes • Unformatted and formatted I/O operations • Use of manipulators	20	10	59
5	**Working with Files,** **Exception handling, Introduction to Template STL**	• File stream classes • Opening and closing a file • Error handling • File modes • File pointers • Sequential I/O operations • Updating a file (Random access) • Command line arguments • Overview of Exception Handling • Need for Exception Handling	20	12	69

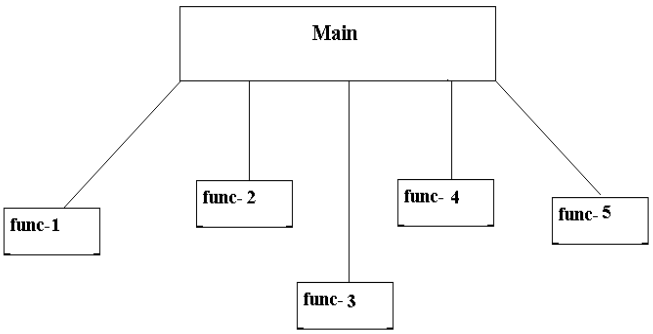
		• various components of exception handling • Introduction to templates • Class templates • Function templates • Member function templates • Overloading of template function • Non-type Template argument • Primary and Partial Specialization • Introduction to STL • Overview of iterators, containers			
Total			100	60	

Unit 1

Principles of Object Oriented Programming Tokens, expressions and control statements

Procedure oriented programming:-(POP)

- When we do conventional programming in high level languages like COBOL, FORTRAN or C. it is known as POP.
- In POP the problem will be considered as input, process and output.
- For doing this work we are writing various functions, so that our work can be done.
- We can summarize this thing in following figures as follows.



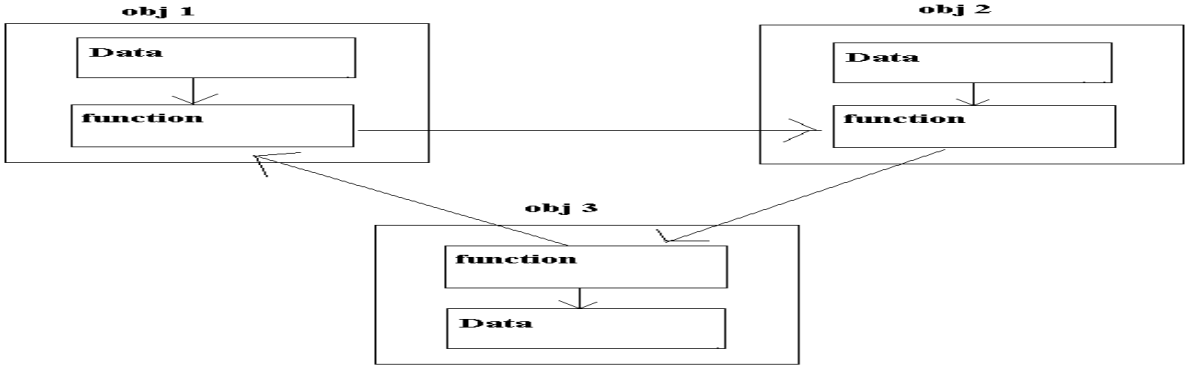
- POP basically consist of instruction that the computer has to follow and if we combine these instructions in a group then that group is known as functions.
- In multifunction program we are using the concept of global declaration so that each and every function can access the global data.
- On the other end each function has its own data also so in a large program it is difficult to identify that which data will affect which function and this will generates error.
- The problem in this approach is it can't handle the real life problems.

Characteristics of POP:-

1) Emphasis on algorithms.
2) Large programs will be divided into small programs called functions.
3) Most of the function uses global declaration of data
4) The data can be travel from one function to another function so there is no data security.
5) These approach uses top-down mechanism.

Object oriented programming :-(OOP)

- The major factor to the invention of the OOP is to remove the drawback of POP.
- OOP treat the data as a critical part and it does not allow the data to move freely in the system.
- OOP will make part of a problem into number of entities called objects and then the data and the functions will be developed around the object only.



- Here the data of the particular objects will communicate with the function of that object only and then the function will communicate with the data using the objects and make inter-function communication.
- The definition of OOP will be "It is an approach that provides a way of modularizing the programs by creating separate memory area and the functions which can be used as templates for creating copies of such objects as a when demanded".
- The main characteristics of OOP are as follows:-
    1) Emphasis on data, not a procedure.
    2) Program will be divided into objects.
    3) Data is hidden so it cannot be access by external functions.
    4) Objects can communicate with each other by using functions.
    5) New objects can be added when it is necessary.
    6) It follows bottom-top approach.

Difference between POP and OOP

NO	POP	OOP
1	Large program will be divided in to several procedures called as **functions**.	Large programs are divided into **objects.**
2	Main part is **procedure**.	Main part is **object**.
3	Top down approach.	Bottom up approach.
4	There is no data security in POP.	There is data security in OOP.
5	Can't handle real life application.	It can handle real life application.
6	Instructions where grouped & call as **Functions.**	Instructions where Grouped & call as **object**.
7	C, FORTRAN, COBOL are the example of POP.	C++, Java is the examples of the OOP.
8	Possibility of error will be more	Possibility of error will be less
9	Addition of new procedure or modification Of exiting. Exiting of procedure is not an easy task.	Addition of new object can be easily done.

Basic concept of OOP

There are some basic concepts in OOP which are as follows:-

1) Objects.
2) Classes.
3) Data abstraction and encapsulation.
4) Polymorphisms.
5) Dynamic binding.
6) Message passing.

1) Objects:-

- Objects are basic runtime entities in OOP.
- It may representation a person, a place, a bank a/c, a table of data, etc. which the program has to handle.
- The objects may represents se define data like vectors, time and list.
- Programming problems will be analyzed in terms of objects and nature of communication between them.

- When the program executed the objects will interact with each other by sending the messages to each other.
- For e.g. if the customer and the bank a/c are two objects then they will communicate with each other whenever the customer will make the transaction.

2) Classes:-

- We know that objects contain the data and they are called to manipulate the data.
- The entire set of the data and code of an object that can be developed using define data types called class.
- In fact objects are the variables of the class.
- Once the class is defined we can create number of objects of that class.
- For e.g., if we create a class named fruits then we can say mango, banana, apple and orange are the objects of the class.

3) Data abstraction and encapsulation:-

- When we combine data and function is a single unit called class then it is called an encapsulation.
- This is the most important feature of OOP.
- Remember that the data is not accessible outside the class and only the functions associated with the class will handle the data.
- This concept is also known as data hiding as information hiding.
- Abstraction refers to the act of representing essential features without including the background details.
- Classes use the concept of abstraction and it will be defined as list of attributes such as size, weight, cost, etc. and the functions to be operated on these attributes.
- The attributes are called data members and the functions are called method or member functions.
- The classes used the concept of the data abstraction i.e. they are called as abstraction data types (ADT).

4) Inheritance:-

- Inheritance is the process by which objects of one class will use the data members and methods of another class.
- It supports the concept of hierarchical classification.
- This concept is most important because we need various objects to use methods and data of various classes.
- In OOP inheritance provide the concept of reusability which means that we can add new features to the existing class without modifying the original class this is possible by deriving a new class from the others class so that the new class has combined features of both the classes.
- In other words we can say that inheritance provides the facility to the programmers to reuse the program i.e. already developed

5) Polymorphism:-

- It is an important concept of OOP.
- This is a Greek term means that to take more work from one objects.
- The objects may change its behavior according to the requirements.
- For e.g. if we say that the operators plus (+) will make addition of two no's and give us the sum.
- On the other hand if we make it possible that the same operators plus will take two string instead of two no's. And gives the joining of two strings that is called as operator overloading.
- On the same way if we have more than one function with the same name and its behavior will be according to the arguments then it is called as function overloading.
- It is known as polymorphism.

6) Dynamic binding:-

- Binding refers to the linking of procedures to the code of runtime.
- Dynamic binding generally used with polymorphism because in function overloading we don't know which function will be called when, so it is called as Dynamic Binding.

7) Message Passing (MP):-

- In OOP when different objects will pass the data between one another then it is called as message passing.
- In OOP we have mainly three steps
    - Create the class and declare the functions.
    - Create the objects from the classes.
    - Transfer the data from one object to another which is called as message passing.

Benefits of OOP:-

OOP provides various benefits to the programmers which are as follows.

1) Through inheritance we can use the concept of reusability.
2) We will develop the programmers which are divided into modules which saves our times.
3) The principle of data hiding use us the security of data.
4) Operators overloading and function overloading gives us multiple thing from one objects.
5) It is easy to make partition of the work.
6) OOP can be easily upgraded to the large system from small system.
7) In OOP we can develop any complex software.

Applications of OOP:-

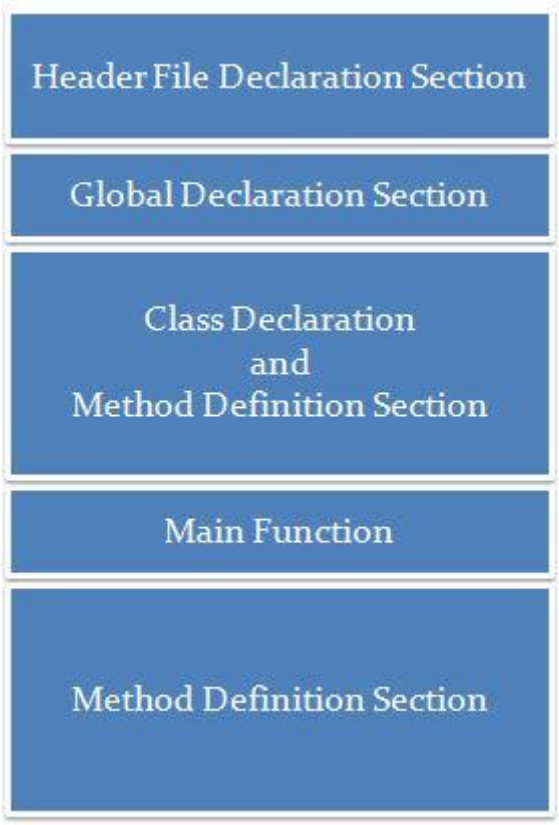
Object oriented programming has many advantages over procedure oriented programming. And that's why it is adopted in variety of area of information technology. Following are the areas where oop can play a great role.

1. **Database:** Low level routines – functionality of all leading RDBMS are developed in C++.
2. **Graphics:** Graphics have changed entire appearance of computer in last decade. Behind this very good eye catching appearance, there is object oriented programming.
3. **Real time application:** Response time is the key factor in real time applications. C++ has a very good above 70% market share in area of real time applications. Majority of security systems, mission critical systems fall in this category.
4. **Embedded programming:** In daily life we use TV, refrigerator, car washing machine, oven etc which are example of embedded programming is product of object oriented programming.
5. **Web based application:** If we focus on web based programming, there are C++, java, .NET and PHP. These all support object oriented programming language.
6. **Mobile application:** For mobile programming we have two languages available in the market. These languages are C++ and J2ME, which are object oriented languages.
7. **Cluster Programming:** In cluster a load of big process is shared among set of processors. This load sharing is done through C++.
8. **AI and expert system:** In artificial intelligence and expert system, decision logic design is done in C++ - the object oriented programming language.
9. **CAD/CAM system:** Computer Aided Design and Computer Aided Manufacturing systems are also developed in C++ and other object oriented programming languages.

What is C++?

- C++ is an OOP.
- It has been developed by Bjarne-stroustrap at AT & T's Ball laboratory USA in 1980.
- Stroustrap was the grate supporter of C language so that he wasn't to combine features of C and OOP in one language so he has developed C++.
- We can say that C++ is the extension of C language.
- Initially it is given the name C with classes but in 1983 the new name C++ was given.
- The idea has come from increment operator ++ which says that increment by one.
- C++ is a superstar of C so if the person knows C language than he easily understood C++.

Structure of C++ Program



1. Header File Section:

In this section, the header files are included that you will need in your program. For example iostream.h is the header file which contains the functions for input output.

Example:

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

2. Global Declaration Section:

In this section all the global variables are declared that are going to be used throughout the progam. This section is optional section i.e. if you need global variables in your program then they are declared here. Here you can include:

-Structure Declaration
-Variable Declaration

Example:

```
# define PI 3.14
int MAX;
```

3. Class Declaration Section:

In this section, you can declare classes you want to use in the program. In the class declaration section, you have to declare and define the functions used by your class. This is also an optional section. You can create a c++ program without a class but to use object oriented features, class is necessary.

4. Function Defination Section:

The functions declared by the class in class declaration section are defined in this section. This is again an option section. If you have declared a class then its function is defined here.

5. Main function:

The main () function is the compulsory part for every C++ program because it is the entry point for the execution of each program.

In main function generally the objects of classes are created. Operation System calls main function automatically when you run the program. So in this section, main () function must be written to run your program.

A sample C++ program:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        cout<<welcome to the first c++ program;
        getch();
}
```

Output:

- **Welcome to the first c++ program:**

In this program, we have included two header files:< iostream.h> is for input /output stream classes. From this header file, we have used the cout function to print message on the screen. The second header file conio.h is for console input/output. From this header file we have used clrscr() (used to clear the screen) and getch() (to get a character from the keyboard) functions.

We haven't used global declaration, class declaration and function definition section as they will be introduced in later chapters. The last section we used is main() function which is compulsory section.

Input/Output Operators

The input operator is used to take input from the keyboard and the output operator is used to print some output on the screen.

Output Operator:

In our sample program, we used cout (Cout is pronounced as "C out") statement to print output. To print something output operator is used with the cout statement. The output operator (<<) is also known as insertion operator because it inserts the text or value of the variable passed with it.

Assume that the statement is as follows:

Cout<<variable_name;

Then the value stored in the v1ariable is transferred to screen by cout object using the insertion operator. If the statement is like:

Cout<<"message";

Then the string enclosed in double quotes is sent to the screen by cout object using the insertion operator.

Input Operator:

As to print some output cout object is used with insertion (Output) operator, to take some input from the keyboard cin (prononunced as " C in") object is used with input operator. Its symbol is >> and is also known as extraction operator because it extracts the value entered through the keyboard. Assume that the statement is as follows:

Cin>> variable_name;

Then the value entered through the keyboard is stored in the variable by cin object using the extraction operator. You can also cascade the extraction operators for multiple inputs. For example:

Cout<<"enter 2 values:";

Cin>>var1 >> var2;

The first value entered by keyboard is stored in var1 and second value is stored in var2.

Introduction of Namespace

To understand the concept of namespace, consider a situation in which you need to create two (or more) classes with same name. For example, in any operating system you can't create two files with same

name in same folder. If you have a file named text.txt in a folder named test. Now you cannot create another file with the same name in this folder. The solution is to store the file in different folder, say demo. The namespace is the same concept. It is used to resolve the name clash problem in your program.

Namespace is just a collection or container of classes. You can add your class, function, and variables etc. code in a namespace to create a useful container of related code. Now you can create another namespace and add classes, functions or variables with same name of the old namespace without any clash.
You can create a namespace with the namespace keyword as shown below:

```
Namespace namespace_name
{
        // statements…..
}
```

To use the class or function from the namespace you have to include a statement saying that you are using that namespace in your program as shown below:
Using namespace namespace_name;
For example:

Using namespace test; The namespace concept will not run under Turbo c++ compiler, so if you want to test namespace in your program, use some advanced C++ editor such as Microsoft visual c++.

Tokens

Tokens are the building blocks of the C++ program. Every single word of the c++ program can be considered as a c++ token. Tokens can be divided in the following categories:

1. Keywords 2. Identifiers 3. Constants 4. Operators 5. Strings

1. KEYWORDS:

Keywords are the reserved identifiers of c++ and they have some specific meaning. You cannot use any keyword name as identifier such as variable, function or class name.
The following table lists keywords of C++. You will see that some of them are common to C language and also have same meaning as in c language.

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while .

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq Your compiler may also include some additional specific reserved keywords.

2. Identifiers:

An identifier is a name given to a variable, function, class, object, structure, union, enumeration a label to identify, it in the program. You can give any name to your identifier but there are some rules to declare an identifier as shown below:

- ➔ Variable name must start with alphabets.
- ➔ After 1st char you can use alphabets or no's or underscore (_).
- ➔ Special symbols and spaces are not allowed in variables names.
- ➔ Keywords are not allowed as variable names.

3. Constants :

As the name suggests, constants are fixed values that should not be changed throughout the program execution. Constants can be of type integer, floating point, characters or strings.
For example:

Integer constants : 123, 99, 10
Floating point constants : 1.23, 40.03, 23.45
Character constants : 'a', 'A', 'x'
String constants : "abc", "string", "computer"

***4. Operators**:*

In the program, you may use operators in expression such as: a= b + c. Here, = and + are operators.

***5. Strings**: Strings are the words you write in the program such as in printing message or in string constants.*

BASIC DATA TYPES

-	Char	1 byte	-128 to +127
-	Int	2 byte	-32768 to +32767
-	Unsigned	2 bytes	0 to 65535
-	Long int	4 bytes	-2147483648 to +214783647
-	Float	4 bytes	$3.4*10^{-38}$ to $3.4*10^{38}$
-	Double	8 bytes	$1.7*10^{-308}$ to $1.7*10^{-308}$
-	Long double	10 bytes	$3.4*10^{-4932}$ to $3.4*10^{4932}$

Above are the basic data types of C++, we also have other two user data types which are as follows:-

STRUCTURE AND CLASSES:

- ➔ In "C" language we have structure and in C++ we have classes.
- ➔ In structure we can define variable only but in class we can define variable and methods.

ENUMERATES DATA TYPES:

- ➔ With enumerates data types we can give constant value to the data.
- ➔ For e.g. if we declare enumerated data types for colors and if we have four colors named red, green, blue and yellow.
- ➔ If we define these colors with enumerated data types then they can be given no's from 0 to 3.
- ➔ Besides these data types we also have derived data types which are arrays, function and pointers.

SYMBOLIC CONSTANT

- ➔ Symbolic constant are the variable where value remains constant through out the program.
- ➔ In "C" language we are declaring the symbolic constant like define program.
- ➔ But here in C++ we have been given a new keyword constant will allow us to declare symbolic constant,
- ➔ For e.g. const int I=10;

OPERATORS IN C++

1) **Mathematical operators:-**
   +, -, * , /, %
2) **Relational operators:-**
   <, >, <=, >=, =, ==, !=
3) **Logical operators:-**
   OR, NOT
4) **Special operators:-**

::	- scope resolution operator
::*	-pointer variable operator
->*	- pointer to member function operator
.	- object to function operator
Delete	- memory variable release operator
End	- new line variable
New	- memory variable allocation
Setw	- width operator

SCOPE RESOLUTION OPERATORS

- Like 'C' language, C++ is a block structure means that each and every thing will be form of blocks.
- In C++ we can declare two different variables with the same name in different blocks.
- For e.g.

```
{                                   //outer block
    Int x= 10;
    {                               //inner block
        Int x=20;
    }
}
```

- Here the above example x will be different values in different blocks but if you want to use global value than we have to use scope resolution operator.

MANIPULATORS

- Manipulators are the operators that are useful for formatting our data.
- The most commonly used manipulation is endl and setw.
- Endl is useful for new line output.
- When we display some data on the screen it will always left handed.
- For e.g.

```
int x=10;
int y=225;
int z=3685;
cout<<y<<endl;
cout<<y<<endl;
cout<<z<<endl;
```

Output is as follows

```
10
225
3685
```

- The above output is not proper because we have to display no's right justify for these purpose C++ has given as a manipulation called setw.

Control structures

It is not necessary that your program should execute only in sequential order. You can change the flow of program execution using control statements. C++ has various control structures for branching and looping. Using branching, you can control the flow of program execution and you can use looping statements to repeatedly execute some statements based on some condition.

Branching statements are also known as decision making statements. C++ provides following decision making statements:

1) if statement. 2)if….else statement 3) Else…. If ladder 4)Nested if statement 5) switch statement

<u>If statement</u>

if statement is a powerful decision making statement and is used to control the flow of execution of statements. It takes the following form.

Syntax:

```
if ( test expression / Condition )
{
        Statement block;
}
        Statement - x;
```

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression is true or false, it transfers the control to a particular statement. This point of program has two paths to follow one for the true condition and other for the false condition. The statement block may be a single statement or a group of statements. if the test expression is true the statement – block will be executed; otherwise the statement – block will be skipped and the execution will jump to the statement – x. remember when the condition is true both the statement – block and the statement –x are executed in sequence.

if - else statement

if – else statement is an extension of the simple if statement. The general form is as below.

SYNTAX:

```
if (test expression)
{
          Statement(s );  - True block
}
else
{
         Statement(s);   - False block
}
          Statement – x (Next Statement);
```

If the test expression is true, then the true block statement(s), immediately following the if statement(s) are executed; otherwise the false block statement(s) are executed. In either case, both the case either true-block or false –block will be executed, not both.

Else if statement (Else If ladder)

In C programming language the else if ladder is a way of putting multiple ifs together when multipath decisions are involved. It is a one of the types of decision making and branching statements. A multipath decision is a chain of if's in which the statement associated with each else is if. The general form of else if ladder is as follows –

SYNTAX:

```
if ( condition 1)
{
      statement - 1;
}
else if (condtion 2)
{
     statement - 2;
}
.
.
else if ( condition N)
{
      statement - n;
 }
 else
 {
        default statment;
 }
          statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top of the ladder to downwards. As soon as a true condition is found, the statement associated with it is executed and the

control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default statement will be executed.

Nested if statement

When a series of decisions are involved we may have to use more than one if-else statements in nested form as follows.

SYNTAX:

```
if ( test condition 1)
{
                    //  If the test condition 1 is TRUE then these it will check
for test condition 2
             if ( test condition 2)
            {
                        //If the test condition 2 is TRUE then these
statements will be executed
                      Test condition 2 True statements;
             }
             else
             {
                                  //If the c test condition 2 is FALSE then these
statements will be executed
                       Test condition 2 False statements;
            }
}
else
{
                     //If the test condition 1 is FALSE then these statements
will be executed
              Test condition 1 False statements;
}
```

If the condition-1 is false, the statement -3 will be executed; otherwise it continues to perform the second test. If the condition -2 is true, the statement-1 will be evaluated; otherwise the statement -2 will be evaluated and then the control is transferred to the statement –x.

SWITCH-CASE:

C has a built in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below.

SYNTAX:

```
switch(expression / Variable )
{      case value  1:
              block -1
              break;
       case value  2:
              block -2
              break;
       ____
       ____
       default:
              default –Block
 }
Statement –x;
```

The expression is an integer expression or character value -1, value-2 is constants or constant expressions are known as case labels. Each of these values should be unique within a switch statement. Block-1, block-2 are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that labels end with colon. When a switch is executed, the value of the expression is successively compared against the values value-1,value-2…….if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

Notice that each group of statement ends with a break statement in order to transfer control out of the switch statement. The break statement is required within each of the first groups in order to prevent the succeeding groups of statement from executing. This last group does not require a break statement since control will automatically be transferred out of the switch statement after the last group has been executed. This last break statement is included; however has a matter of good programming practice. So that it will be present, if another group of statement is added later

Iterative (Looping) Control Statements

If we want to perform certain action for no of times or we want to execute same statement or a group of statement repeatedly then we can use different type of loop structure available in C. Basically there are 3 types of loop structure available in C

(1) For loop (2) While loop (3) Do While Loop

FOR Loop :

The for statement is entry controller that provides a more concise loop control structure. This is most popular loop control.

SYNTAX:

```
for(initialize counter; test counter; increment counter)
{
    statement 1;
    statement 2;
}
```

The execution of the for statement is as follows:

1. Initialization of the control variables is done first, using assignments statement such as I=0 and count=0. The variables I and count are known as loop controls.
2. The value of the control variables is tested using the test condition. The test condition is relational expression, such as I>0 or I<10 that determines when the loop is terminated and the execution continues with statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement either evaluating the last statement is the loop. Now, the control variable is incremented using and assignment statement such as I=I+1 and if the new value of the control is satisfied under test condition, then the body of the loop is executed. This process continues till the value of the control variable fails to satisfy the test-condition.

EXAMPLE:

int i;
for(i=1; i<=10;i++)
  cout<<i;
 	
Output is:12345678910
	int i;
	for(i=10; i>=1;i--)
	  cout<<i;
	
	Output is:10987654321

That the initialization, testing and incrimination of loop counter is done in the for statement itself. Instead of i=i+1, the statement i++ or i+=1 can also be used. Since there is only one statement in the body of the for loop, the pair of braces have been dropped. The for statement allows for negative increments. This loop is also executed 10 times. But the output would be from 10 to 1 instead of 1 to 10. Note that, braces are optional when the body of the loop contains only one statement.

We can also write for loop this way.

int i;
for(i=1; i<=10;)
{
Cout<<i;
i=i+1;
}
Output is:12345678910
	int i=1;
	for(; i<=10;)
	{
	Cout<<i;
	i=i+1;
	}
	Output is:12345678910

Here the increment is done within the body of the for loop and not in the for statement. Note that inspite of this the semicolon after the condition is necessary. In another example neither initialization, nor the incrementation is done in the for statement, but still two semicolon are required.

Additional features of for loop :

Multiple initializations:

The for loop in C has several capabilities that are not found in the other loop constructs. For example, more than one variable can be initialized at a time in the for statement.

This is perfectly valid. The multiple arguments in the increment section are separated by commas.

```
int i,j;
    for(i=1,j=2; j<=10;j++)
```

Infinite loop:

infinite loop. Such loops can be broken using break or go-to statements in the loop. We can set up time delay loop using the null statement as follow:

for(j=1000;j>0;j--) {}

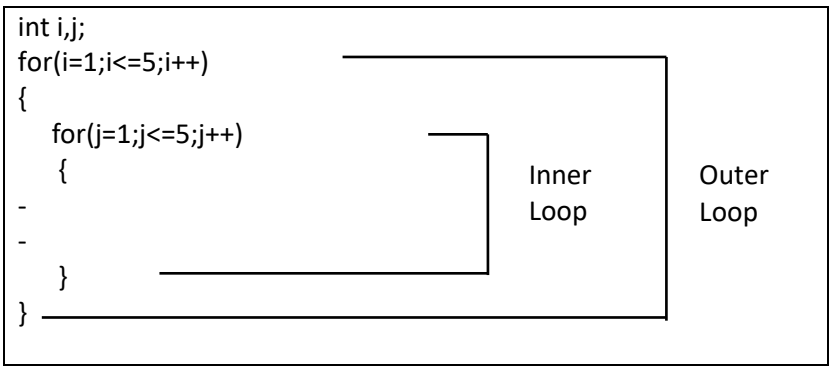
The loop executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as null statement. This can also be written as :

for(I=1000; I>0; I--);

This implies that C compiler will not give an error message. If we place a semicolon by mistake at the end of a for statement. The semicolon will be considered as a null statement and the program may produce some nonsense.

Nested loop:

Loops can be nested (i.e. embedded) one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential that one loop be completely embedded within the other. There can be no overlap. Also each loop must be controlled by a different index.



The way for loops have been nested here similarly, two while loops can be nested. Not only this, a for loop can occur within a while loop or a while loop within a for loop.

WHILE Loop:

The while statement is used to carry out looping operations. The general form of the while statement is as below.

SYNTAX:

```
Initialization;
while(test condition)
{
   Statement 1;
   Statement 2;
   increment/decrement;
}
```

The enclose statements within two braces will be executed repeatedly as long as the expression evaluated as true. When the expression is evaluate a false or when condition will be false then it will come out from the loop and stop the execution of that loop. Initialization statement initializes some memory variable with some value. Increment or decrement operator increase or decrease the value of operator is use by expression. This statement can be simple of compound, though it is typically a compound statement, it must include some features, which eventually offers the value of expression, thus provides a stopping condition for the loop. While loop construct provides an entry… controlled loop.

The practice, the included expression is usually a logical expression that is either true or false. (Remember that true corresponds to non-zero value and false corresponds to zero value). Thus the statement will continue to execute as long as the logical expression is true.

The part of while statement, which contains the statement, is called the body of the loop. Body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

EXAMPLE:

Suppose we want to display the consecutive digits 1,2,3,…10. This can be accomplished with the following program.

```
#include<iostream.h>
#include<conio.h>
Void main()
{
   int i=1;
   while(i<=10)
   {
     Cout<<i;
     i++;
    }
}
Output:12345678910
```

Initially, digit is assigned a value of 1. The while loop then displays the current value of digit, increases its value by 1 and then repeats the cycle, until the value of the loop will be repeated by 10 times, resulting in 10 consecutive numbers of output.

DO WHILE Loop :

The do while loop looks like this.

SYNTAX:

```
Initialization;
do
{
   Statement 1;
   Statement 2;
   increment/decrement;
} while(test condition);
```

The enclosed statement within 2 braces will be executed repeatedly as the value of expression is not zero. Notice that statement will always be executed at least once. Since the test for repetition does not occur until the end of the first pass through the loop. The statement can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature which eventually alters the value of expression so that the looping action can terminate. Since the expression is evaluated at the bottom of the loop, the do…while loop construct provides an exit… controlled loop and therefore the body of the loop is always executed at least once. In practice, expression is usually a logical test expression, which is either true(with a non zero value) or a false(with a value of 0). The included statement will be repeated (i.e. another pass will be made through the loop) if the logical expression is true. For most applications it is more natural to test for continuation of a loop at the beginning rather than at the end of the loop for this reason, the do…while statement is use less frequently than while statement.

EXAMPLE:

```
#include<iostream.h>
 int main()
{
   int i=1;
   do
   {
     cout<<i;
     i++;
    } while(i<=10);
   return 0;
}
Output:12345678910
```

The digit is initially assigned a value of 1. The do while loop displays the current value of digit. Increases its value by 1, and then tests to see if the current value of digit exceeds 10. If so loop terminates; otherwise the loop continues, using the new value of digit.

FUNCTIONS IN C++

INTRODUCTION:

Functions can be considered as the building blocks of C++ program. In general, a function is nothing but a block of code or statements that perform some specific task. For example the printf() function of c language performs the function of printing output, scanf() function scans the input from keyboard, clrscr() function clears the screen etc.

There are two main purposes of writing a function:

1. To divide a large and complex program into blocks (functions) so that it becomes more readable.
2. Increase the reusability of program. Once a function is defined, you can use it number of times without rewriting its code.

Working of functions:

function has 3 parts:

1. Function declaration (Also known as Function Prototype)
2. Function Definition
3. Function Call

Function declaration specifies the structure of the function. It tells the compiler the name of the function, number and types of arguments and return type of function.

Function definition, as its name suggests, defines what the function will do when it will be called.

Function call is the statement which calls the function. When a function is called it transfers the control to the function definition and executes each statement of the function body (function definition). You can call a function as many times as you want.

Inline functions

→One of the main objectives of OOP is reusability.
→This is useful to save the memory space.
→This is possible when it is necessary to call a member repeatedly.
→Remember that when you call any function than it is always take a lot of extra time fir executing a series of instruction, jumping to the function saving the registration pushing the argument into the stack and returning the vale to the calling function.
→For this purpose when we are using "c" language we are defining macros of we can say pre-processor directives like #define.
→Here in C++ does not support the concept of pre-processor, so C++ has given us a different solution which is called as inline function.
→In line function will work just like a simple UDF but the specialty of inline function is when we call a simple UDF then each and every time the UDF and its argument will be loaded in the memory but if we declare the UDF as inline, then the same memory location will be used so it will save our memory space as well as gives us better speed.

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
inlineint mul(int,int);
void main()
{
        int a,b,c,x,y,z;
        a=10;
        b=20;
        x=30;
        y=40;
        c=mul(a,b);
        z=mul(x,y);
        cout<<"c="<<c<<endl;
        cout<<"z="<<z<<endl;
        getch();
}
int mul(int p,int q)
{
        return(p*q);
}
```

Default Arguments:-

→C++ allows us to create a function in which when we call the function without giving all its arguments. This concept is known as default argument.
→We can define default arguments at the time of declaration of function.
→For e.g. float amount(float, int)
float r=12;
→Here in the above statement r is the default value, so when you call the above function you can call it like:-
si=amount (5000, 7)
→In the above call r is missing, but r will take a default value i.e. 12.
→Now if we want to change the value of r at run time as, the time of calling function we can also do that as si=amount (5000, 7, 9)
→Here the function will not take default argument i.e. 12, instead of that it will take r=9.
→The most important thing that we have to remember is when we want to use default argument then all the default argument must be in the end of proto-type.

1) *int add(int i, int t, int k=10) -valid*
2) *int add(int i, int j=5 ,int k-15)-valid*
3) *int add(int i=10, int j=4, int k=15)-valid*
4) *int add(int i=5, int j, int k)-invalid*
5) *int add(int i=5, int j, int k=10)-invalid*

METHOD OVERLOADING OR FUNCTION OVERLOADING:-

→Overloading refers to the use of same thing for different purpose.
→C++ allows us overloading of functions which means that we can declare two or more functions with same name but for different purpose with different thing of arguments or returns type of data type.
→This is known as function overloading or method overloading of function polymorphism.
→**Examples:-**

- ***int add(int x, int y);***
- ***float add(float a , float b);***
- ***int add(int x, int y, int z);***
- ***float add(int x, int y, int z);***
- ***float add(int x, int y, float a);***
- ***float add(int x, float a, float b);***

→In the above example, if we create all the six functions in the single program than when we run the program at that time according to the argument the function will be called.

PROGRAM

```
#include<iostream.h>
#include<conio.h>
float mul(float,float);
int mul(int,int);
void main()
{
        int a,b,c;
        float x,y,z;
        clrscr();
        cout<<"enter two integers"<<endl;
        cin>>a>>b;
        cout<<"enter two floats"<<endl;
        cin>>x>>y;
        c=mul(a,b);
        z=mul(x,y);
        cout<<"multiplication of two integers are=>";
        cout<<c<<endl;
        cout<<"multiplication of two floats are=>";
        cout<<z;
        getch();
}
int mul(int a,int b)
{
        int p;
        p=a*b;
        return(p);
}
float mul(float x,float y)
{
        float q;
        q=x*y;
```

```
        return(q);
}
```

VARIOUS TYPES OF FUNCTIONS:-

1) Ceil(x):- Returns the near integer more than the given value of x
   Examples:- ceil(x):- it returns the values -9
2) Cos(x):- Returns the cos value for given x.
3) Exp(x):- return the exponential value for given x, where e=2.7 & function return $e^{x.}$
4) Floor(x):- Return the near integer less than the given no.
5) Log(x):- Returns the log natural for given x.
6) Log_{10}(x):- Returns the log lo the user function x for the base 10.
7) Pow(x):- Returns x rest to power y. (x^y)
8) Sin(x):- Returns the sign value of x.
9) Sqrt(x):- Return the square root of x.
10) Tan(x):- Return the tangent value of x.

Adding C functions in turbo C++

In turbo c++, you can use C functions also. You should have the knowledge of which header file consists the functions. For example, to use functions like printf() and scanf() you have to include stdio.h header file the following example uses stdio.h header file to use printf() function.

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>  //C header file
Int main()
{
    clrscr();
    printf("This is C function");
    cout<<"\n This is C++ function";
    getch();
    return 0;
}
```

OUTPUT:

```
This is c function
This is C++ function
```

Unit 2

Classes and Objects, Constructor and Destructor

WHAT IS CLASS?

→The class is a way to bind the data and its associated function's together. It allows the data (and functions) to be hidden, if necessary, from external use. Generally, a class, we of the class definition an as two parts

- Class declaration.
- Class function definitions.

→The class declaration describes the type and scope of it's members the class function definitions describes how the class functions are implemented.

→The general form of class declaration is:

```
class (class.name)
{
    private:
    variable declarations;
    function declarations;
    public:
    variable!= function declarations
};
```

→The class declaration is similar to a structure declaration of C language. The keyword class define specify the match class, the class coding contains the declaration of variable that function which are collectivity as far as members.

→There are usually groped under two sections the class member that is declare under private can be access by the class only keyword the member declare the public section can be access how the class also.

→The variable declaration inside the class are known as data members, while the functions are called as member functions as method communication of both this is known as an encapsulation.

What is Object? OR Creation of objects.

→When we declare class and if we have various members that are variable and function can be accessed only through the object of the class.

→To declare an object first we have to create a class and then in main function we have to create object of the class.

Examples

```
Class ABC ();
    {
        Private:

        Public:
    };
    Void main ()
    {

        ABC p;

    }
```

→Here the above examples name of the class in ABC & p is its objects.

Example:

A class example with function declaration inside the class.

```
#include<iostream.h>
#include<conio.h>
class abc
{
        private:
        int a,b,c;
        public:
        void getdata (void)
        {
                cout<<"enter two no:";
                cin>>a;
                cin>>b;
        }
        void process(void)
        {
                c=a*b;
        }
        void putdata (void)
        {
                cout<<"ans is   : "<<c;
        }
};
void main()
{
        abc p;
        clrscr();
        p.getdata();
        p.process();
        p.putdata();
        getch();
}
```

A class example with function declaration with outside the class.

```
#include<iostream.h>
#include<conio.h>
class abc
{
        private:
                int a,b,c;
        public:
                void getdata(void);
                void process(void);
                void putdata(void);
};
                void abc::getdata(void)
                {
                        cout<<"Enter the two no-->";
                        cin>>a>>b;
                }
                void abc :: process(void)
                {
```

```
                    c=a*b;
            }
            void abc :: putdata(void)
            {
                    cout<<"ans==>"<<c<<endl;
            }

    void main()
    {
            abc p;
            clrscr();
            p.getdata();
            p.process();
            p.putdata();
            getch();
    }
```

Access Specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in c++ programming language private, public and protected.

1) *A private member within a class denotes that only members of the same class have accessibility. The private member is inaccessible from outside the class.*
2) *Public members are accessible from outside the class.*
3) *A protected access specifier is a stage between private and public access. If member functions defined in a class are protected, they cannot be accessed from outside the class but can be accessed from the derived class.*

When defining access specifiers, the programmer must use the keywords: private, public or protected when needed, followed by a semicolon and then define the data and member functions under it.

```
Class test
{
        Private:
int x, y;
Public:
Void sum()
{
 ……..
……
}
};
```

In the code above, the member x and y are defined as private access specifiers. The member function sum is defined as a public access specifier.

Nesting of member function

→We know that member function can be called only with the help of object of the particular class.
→But sometimes, we need that a member function of a class can be called from another function of the same class.
→This is known as Nesting of member function.

Program:

```
#include<iostream.h>
#include<conio.h>
class abc
{
        private:
                int x,y;
        public:
                void getdata(void);
                void display(void);
                void largest(void);
};
void abc::getdata(void)
{
        cout<<"Enter the no  :"<<endl;
        cin>>x>>y;
}
void abc::display(void)
{
        cout<<"Largest value is  :"<<endl;
}
void abc::largest(void)
{
        if(x>y)
        {
                cout<<x;
        }
        else
        {
                cout<<y;
        }
}
void main()
{
        abc p;
        clrscr();
        p.getdata();
        p.display();
        p.largest();
        getch();
}
```

Private member function

→In normal form, we are declaring variables under private section & function under public section but in some situations, we need to restrict the class object from accessing the member function by the object.

→This is called as a “Concept of data hiding”.

→In C++, it is possible when we declare member function under private section.

→Remember that the private member function can be called only by another member function of the same class inside of calling it by the object of the class using of .(dot) operator.

→Example

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                int x;
                void read(void);

        public:
                void update(void);
};
void test :: read(void)
{
        cout<<"Enter a number :";
        cin>>x;
}
void test :: update(void)
{
        read();
        x=x+10;
        cout<<"Value is  :"<<x;
}
int main()
{
        test a;
        clrscr();
        a.update();
        getch();
        return 0;
}
```

→In the above example read function has been declare under the private section so it can't be directly called by the object of class means that if P is an object of the class we are not allowed to called the function like **P.read**

→Now, if you want to call read function then we have to complusary use the concept of nesting of function means that we have to called read function through update function.

Array within the class

→Just like normal C & C++ program, we can use arrays within the class also.

→We have to declare array variable just like normal variable inside the private section.

Memory allocation for objects

→When we declare any object for the particular class at that time the object will occupy by the memory space specify the class but this statement is not 100% true.

→ Because memory allocation will be done two times:

1) Once when the class has been created all the function declared inside the class will occupy the memory space.
2) In second space when you declare object of the class memory space will be occupy for variables declare inside the class.

Static data member

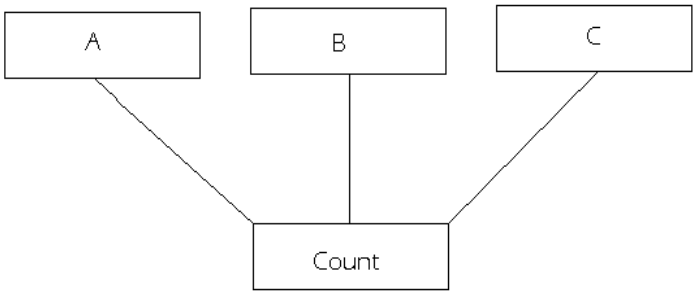
→There is a main two parts:

1) Static variables
2) Static functions

→We can declare any variable of the class as Static.

→When we declare any variables as static it has to following the properties:

1) It initialize to zero when the first object of the class has been created.
   No another initialization will be possible.
2) Only one copy of variable has been created and it will be shared by all the objects.
3) Its lifetime will be throughout the program.



→In the above figure variables count has been declared as static their will be only one copy of that variables will be created and it is said by all the member that is A, B & C.

Example:

```
#include<iostream.h>
#include<conio.h>
class item
{
        private:
                staticint count;
                int no;
        public:
                void getdata(int a);
                void getcount(void);
};
void item::getdata(int a)
{
        no=a;
        count++;
}
void item::getcount(void)
{
        cout<<"total==>"<<count<<endl;
}
int item::count;
void main()
{
        item a,b,c;
        clrscr();
        a.getcount();
```

```
        b.getcount();
        c.getcount();
        a.getdata(1);
        b.getdata(2);
        c.getdata(3);
        a.getcount();
        b.getcount();
        c.getcount();
        getch();
}
```

Static member function

→Like static member variables e can have static member function.

→When declare any member function is static, it haves following properties:

1) A static function can only access to other static member of the same class only.
2) Static member function can't be called using. (Dot) operator but you have call it with scope resolution (::) operator.

Program :

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                int code;
                staticint count;
        public:
                void setcode(void)
                {
                        count++;
                        code=count;
                }
                void showcode(void)
                {
                         cout<<"code="<<code<<endl;
                }
                staticvoid showcount(void)
                {
                        cout<<"count="<<count<<endl;
                }
};
int test::count;
void main()
{
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();
        test t3;
        t3.setcode();
```

```
        test::showcount();
        t1.showcode();
        t2.showcode();
        t3.showcode();
        getch();
}
```

Array of objects

→We know that array can be of any general data type or structure but here in C++, we can declare array of class object also for e.g.

```
Class test
{
        - -
        - -       |
};
```

→Now in the main function if we declare the object as follows than it becomes array of object.

→For e.g.

```
Test t [5];
```

→Here the array of % objects has been created having range t[0] to t[4].

Object as function argument

→Like regular data type, we can also pass any object as argument to any function.

→This is possible in two different ways:

1) A copy of 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, and 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.

Program

```
#include<iostream.h>
#include<conio.h>
class time
{
        private:
                int h,m;
        public:
                void gettime(int hh,int mm);
                void puttime(void);
                void sum(time,time);
};
void time::gettime(int hh,int mm)
{
        h=hh;
        m=mm;
}
```

```
void time::puttime(void)
{
        cout<<"houres=="<<h;
        cout<<"minuts=="<<m;
}
void time::sum(time t1,time t2)
{
        m=t1.m+t2.m;
        h=m/60;
        m=m%60;
        h=h+t1.h+t2.h;
}
void main()
{
        clrscr();
        time t1,t2,t3;
        t1.gettime(2,45);
        t2.gettime(3,30);
        t3.sum(t1,t2);
        cout<<;
        t1.puttime();
        cout<<"t2=";
        t2.puttime();
        cout<<"t3=";
        t3.puttime();
        getch();
}
```

Friend function

When a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non member functions and external classes. For handling such cases, the concept of Friend functionsis a s useful tool. A c++ friend functions are special functions which can access the private members of class. The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in c++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function defination.
- It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
class sample
{
```

```
        int a;
        int b;
        public:
                void setvalue()
                {
                        a=25;
                        b=40;
                }
                friend float mean(sample s);
};
float mean(sample s)
{
        return (s.a+s.b)/2.0;
}
void main()
{
        sample x; //object x
        clrscr();
        x.setvalue();
        cout<<"mean value ="<<mean(x)<<"\n";
        getch();
}
```

Returning object

→As any method can receive object as arguments in the same way function can returns objects also.

Program

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                int x,y;
        public:
                void setdata(int a,int b)
                {
                        x=a;
                        y=b;
                }
                friend test sum(test,test);
                void show(test);
};
test sum(test t1,test t2)
{
        test t3;
        t3.x=t1.x+t2.x;
        t3.y=t1.y+t2.y;
        return(t3);
}
void test::show(test t)
{
        cout<<t.x<<endl<<t.y<<endl;
}
void main()
```

```
{
        test p,q,r;
        clrscr();
        p.setdata(5,10);
        q.setdata(50,100);
        r=sum(p,q);
        cout<<"p=";
        p.show(p);
        cout<<"q=";
        q.show(q);
        cout<<"r=";
        r.show(r);
        getch();
}
```

Pointer to member

→Just like the regular variables, it is possible to tale a member of the class, find out its address & assign it to a pointer.

→A class member pointer can be declared using (::*) operator with the help of class name.

→For e.g.

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

→Now we can define pointer to the class member M in following way

```
Int a ::*p=&a :: m;
```

→Here we have declared P as a pointer for the class member M but if we write like:

```
Int *p=&m;
```

→The above statement will not work because it requires access to the class while that access is available only with the class so whenever we want to access any member of the class we have to use scope resolution operator.

Local classes

→When we declare any class inside the function then this is called as local class.

→Local classes can have access to its local variables declared in the function inside which we have declared the class.

→Local class objects can also use global variables with the help of scope resolution operator.

```
void f();
 int main()
{
  f();
  return 0;
}

void f()
```

```
{
  class myclass {
    int i;
  public:
    void put_i(int n) { i=n; }
    int get_i() { return i; }
  } ob;

  ob.put_i(10);
  cout << ob.get_i();
}
```

Nested Classes:

You can create nested class by defining a class inside another class. The class defined inside the class is known as the inner class and the class in which class is defined is known as outer class:

General form:

```
Class OuterClass
{
        Class InnerClass
        {
                //code
        };
}
```

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
Class Outer
{
        Public:
                Class Inner()
                {
                Public:
                Void showInner()
                {
                        Cout<<"inner class"<<endl;
                }
};
Void showOuter()
{
        Cout<<"outer class"<<endl;
        Inner i;
        i.showInner();
}
};
Void main()
{
        Clrscr();
```

```
        Outer o1;
        O1.showOuter();
        Getch();
        return 0 ;
};
```

CONSTRUCTOR & DESTRUCTOR

Introduction of Constructor

→In a class when we declare a function than we call that function by using the object of he class.

→This function must be invoked by the specific object to run it but up till now we don't have any function which runs automatically without calling it.

→Our regular function has problem also that is, it doesn't allows us to initialize our private variables.

→For these purpose C++ provides us a special member function called Constructor.

→The constructor is special function because it has the same name as the class has means that same name of the constructor function.

→Normally constructor is useful to initialize the variables of the class.

→It is called constructor because it constructs the class objects

→This function posse some special characteristic which is as follows:

1) Constructor is to be declared inside the public section only.
2) They are invoked automatically when the objects of the class are created.
3) They don't return any value so they have no return data types (not even void).
4) Like C++ function they can have arguments.
5) They can't be virtual.
6) We can't able to refer to the address of constructor.

Program:

```
#include<iostream.h>
#include<conio.h>
class abc
{
        private:
                int x,y;
        public:
                abc();
                void show(void);
};
void abc::show(void)
{
        cout<<"x="<<x;
        cout<<"y="<<y;
}
abc::abc()
{
        x=0;y=0;
}
void main()
{
        abc p;
        clrscr();
        p.show();
        getch();
```

```
}
```

Parameterized constructor

→It doesn't necessary to initialize all the variables with 0(zero) but sometimes we also need to initialize the variables of different objects with different values.

→For these purpose C++ allows us to develop a constructor having parameters pass to it.

→Remember that whenever you create parameterized constructor it will never call automatically because we have to pass arguments with each & every object created by us.

Program

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                int x,y;
        public:
                test(int,int);
                void display(void);
};
void test::display(void)
{
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
}
void test::test(int m,int n)
{
        x=m;
        y=n;
}
void main()
{
        test t1(5,10);
        test t2=test(10,20);
        t1.display();
        t2.display();
        getch();

}
```

constructor overloading

→Just like normal function overloading, we can also have constructor overloading.

→Here, we have more than one function with the same name as the class has.

→Because we know that constructor have the same name as the class has.

->When we develop or use constructor overloading the arguments will be different for all the constructors

Program:

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                int x,y;
        public:
```

```
                test()
                {
                }
                test(int a)
                {
                        x=y=a;
                }
                test(int a,int b)
                {
                        x=a;y=b;
                }
                friend test sum(test,test);
                friendvoid show(test);
};
test sum(test t1,test t2)
{
        test t3;
        t3.x=t1.x+t2.x;
        t3.y=t1.y+t2.y;
        return(t3);
}
void show(test t)
{
        cout<<t.x<<endl<<t.y<<endl;
}
void main()
{       test p(10,20);
        test q(20);
        test r;
        r=sum(p,q);
        clrscr();
        cout<<"p=";
        show(p);
        cout<<"q=";
        show(q);
        cout<<"r=";
        show(r);
        getch();
}
```

Constructor with default argument

Like normal functions of C++, you can also set default arguments in constructors also. The same rules are applied to the constructors for default arguments as for the functions. The constructor will consider the default argument if no values is specified for it.

For example:

```
Class Interest
{
        Double p, r;
        Int n;
        Public:
                Interest (double p, int n, double r=0.12);
```

```
};
```

Here, at the time of creating object of interest class, if you do not specify value of r(rate of interest), it will consider it 0.12% as it is set as default argument, and if you specify all the values, the specified value for r will be considered.

Copy constructor

→ A Copy constructor is useful to initialize one object with reference to another object.

→It means that in this type of constructor a reference of one object will be passed to initialized the another object.

→In simple words we can say that Copy constructor is useful to initialize the member of the class for one object with the help of another object.

→Remember that a Copy constructor will always have the reference of objects as arguments & it must be declared in the public section.

Program:

```
#include<iostream.h>
#include<conio.h>
class code
{
        private:
                int id;
        public:
                code()
                {
                }
                code(int a)
                {
                        id=a;
                }
                code(code &x)
                {
                        id=x.id;
                }
                void display(void)
                {
                        cout<<"id="<<id<<endl;
                }
};
void main()
{
        clrscr();
        code p(100);
        code q(p);
        code r=p;
        cout<<"p=";
        p.display();
        cout<<"q=";
        q.display();
        cout<<"r=";
        r.display();
        getch();
}
```

Dynamic constructor

In the dynamic constructor the memory is allocated to the object dynamically at the time of creation of objects. It will save the memory as only the required amount of memory is allocated to the objects. The new operator is used to allocate memory to the objects. In the following example, we have created two constructors: one is default constructor and in the other constructor we have allocated memory as per the size of string passed to it and one additional space for null character

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
class string1

{

        char *name;
        int length;
        public:
        string1()
        {
                length=0;
                name=new char[length+1];
        }
        string1(char *s)
        {
                length=strlen(s);
                name=new char[length+1];
                strcpy(name,s);
        }
        void display(void)
        {
                cout<<name <<endl;
        }
        void join(string1 &a,string1 &b)
};
        void string1::join(string1 &a,string1 &b)
        {
                length=a.length+b.length;
                delete name;
                name=new char[length+1];
                strcpy(name,a.name);
                strcat(name,b.name);
        }
void main()
{

        char *first="joseph";
        string1 name1(first),name2("louis"),name3("langrange"),s1,s2;
        s1.join(name1,name2);
        s2.join(s1,name3);
```

```
        clrscr();
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();
        getch();
}
```

MIL and Advantages of MIL

The MIL (Member initialization List) is a way by which you can initialize the member variables in the constructor. The list of members to be initialize is written with constructor separated by comma followed by a colon.

Syntax:

```
        Constructor(arg1,  arg2, ….) : member_variable1(value), member_variable2(value),…
{
        // other statements
}
```

Here, in the parenthesis near member variable you can also pass expressions:

Example

```
        Number (int a, int b) : x (a + b), y (b*2)
        {
                //other statements.
        }
```

Advantages of using MIL:

There are definitely some advantages of using MIL rather than the normal assignments we do such as:

```
Constructor(int a, int b, int c)
{
        x=a;
        y=b;
        z=c;
}
```

1) The member initialization list is more efficient than the normal assignments.
2) It actually initializes the member variables because, the assignment-version constructor first calls default constructor to initialize the member variables.
3) So all the work performed by the default constructor is wasted and done again.
4) You can also use expressions in member initialization list which will save your code and also the execution time.

Example:

```
#include<iostream.h>
#include<conio.h>
class Test
  {
        Int a, b;
        public:
                Test(int x, int y) : a(x), b(y) //member initialization List
```

```
        {
                Cout<<"constructor using MIL";
        }
        void display()
        {
                cout<<"a="<<a<<endl;
                cout<<"b="<<b;
        }
};
void main()
{
        Clrscr();
        Test t(11,22);
        t.display();
        getch();
}
```

Destructors

A destructor is also a special kind of member function which is used to destroy the object created by constructor. It is special because like constructor, it has also same name as the classname. The destructor is written by specifying a tilde(**~**) sign before its name.

For example, the destructor for class test can be written as:

```
~test()
{
        //code….
}
```

Following are some characteristics of destructor:

1) It has same name as the class name
2) It starts with the tilde(~) sign.
3) It does not take any arguments
4) It does not return any value.
5) It is called automatically when an object goes out of scope.
6) It releases the memory allocated to the object by constructor.
7) Destructors cannot be overloaded.

Importance of Destructor:

When you use constructors to create objects. It allocates memory to those objects. Now as the new objects are created more and more memory is allocated to the objects. At some point these constructors may not be in use i.e. in the scope but still they have occupied some memory. In some systems the memory is very important so we have to take care about memory management.

In C++, destructors are the solution to this problem. When an object goes out of the scope, destructor is called automatically and it releases the memory allocated by the object. Thus it saves the unnecessary memory allocated to the objects which are not in the scope.

The destructor destroys the object by releasing the memory allocated by the constructor if in the constructor, the memory is allocated by new keyword, it should be deleted by delete keyword in destructor.

For example:

```
test()
{
```

```
        a= new char[len +1];//allocating memory  using new keyword
}
~test()
{
        delete a;          //releasing memory using delete keyword
}
```

Following example demonstrates constructor and destructor:

```
#include<iostream.h>
#include<conio.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--;
        }
};
void main()
{
        clrscr();
        cout<<"\n\n enter Main\n";
        alpha a1,a2,a3,a4;
        {
                cout<<"\n\n enter block1\n";
                alpha a5;
        }
        {
        cout<<"\n\n enter block2\n";
        alpha a6;
        }
        cout<<"\n reenter block main\n";
        getch();
}
```

Unit 3

Operator overloading and type conversion, Inheritance

→Operator overloading is one of the most existing feature of C++.
→It is an important technique that enhances the power of extensibility of C++.
→Since begin of C++, we tries to make user define data type that we have just link built in data type.
→In C or C++ we can make addition of two integers or float. But
here with the help of operator overloading C++ allows as to make addition of two objects also.
→In simple words we can say that operator overloading mean to give operator an extra work in which the operator will perfume its regular duty, with some extra duty also.
→In C++ we are allowed to overloading almost all the operator. Excepts the following (operator that are not allowed to be overloaded)

1) Class member access operator (.)
2) Scope desolating operator (::)
3) Size of operator
4) Conditional operator-tannery operator (? :)

→Remember that when all overloading any operator its original meaning must not changing means that we can not overload plus(+) operator to make subtraction.

How to defined operator overloading

→To define an addition task to an operator, we have declared a special funneling called operator function which describes the task.
→The syntaxes for defined the operator overloading function will be as following.

```
Return type class name::operator op (arguments)
{
        - - - - - - -
        - - - - - - -
}
int operator+(int a, int b)
{
        Return a+b;
}
```

→In the above syntaxes op will be replaced by the operator which you want to overload.
→Here the operator function must be either member function or friend function. And it must be declared in the class.
→Just like our normal function. This function can also declare and called through the member of class.

Overloading unary operator

→In mathematic we have only one operator which will work as unary operator as well as binary operator as well a binary operator. That is minus.(-)
→When the minus (-) will be used as unary operator. It has just one operator but when it task two operators. It will work as binary operator
→When we used minus (-) as unary operator it will change the sing of the operator from positive to negative or negative to positive.
→At the same time it will als0 change the value but when the minus (-) operator will work as binary operator than it will subtract operator from the first operator.

Program:

```
#include<iostream.h>
```

```
#include<conio.h>
class abc
{
        private:
                int x,y,z;
        public:
                void getdata(int a,int b,int c);
                void display(void);
                voidoperator-();
};
void abc::getdata(int a,int b,int c)
{
        x=a;
        y=b;
        z=c;
}
void abc::display(void)
{
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
        cout<<"z="<<z<<endl;
}
void abc::operator-()
{
        x=-x;
        y=-y;
        z=-z;
}
void main()
{
        clrscr();
        abc p;
        p.getdata(10,-20,30);
        p.display();
        -p;
        p.display();
        getch();
}
```

Overloading Binary operator

→ The mechanisms of overloading the unary operator will work same as overloading the binary operator means that we can overload the binary operator with the same concept of same unary operator

Program:

```
#include<iostream.h>
#include<conio.h>
class test
{
        private:
                float x,y,t;
        public:
```

```
                test()
                {
                }
                test(float a,float b)
                {
                        x=a;
                        y=b;
                }
                test operator+(test);
                void display();
};
test test :: operator+(test)
{
        test temp;
        temp.x=x+t.x;
        temp.y=y+t.y;
        return(temp);
}
void test :: display(void)
{
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
}
void main()
{
        clrscr();
        test t1,t2,t3;
        t1=test(2.5,3.5);
        t2=test(5.5,6.5);
        t3=t1+t2;
        cout<<"t1=";
        t1.display();
        cout<<"t2=";
        t2.display();
        cout<<"t3=";
        t3.display();
        getch();
}
```

Overloading the binary function using FRIEND function

→ In the previous example we have written like t3=t1+t2 which is evaluated to t3=operator + (t1+t2).
→ In some situation we need that we want the make the use of operator overloading when are passing two arguments in which both the arguments are not possible to be the objects of the class.
→It may be one object & another will be normal variable or value now in some situation, if you call the function or if you overload the binary operator with the change in the sequence of the operand than C++ will not able to do the work.

Example:

T3=t1+5 or t3=5+t1 refers to the different notation in such cash you have to use *Friend* function to overload the operator because when you overload the operator with the Friend function then it requires only one argument which is the appropriate.

→The class object for calling the function in above example we are overloading two operator that multiply i/p or o/p operator that is ***<<&>>***

Program:

```
#include<iostream.h>
#include<conio.h>
Class Numbers
{
        float a, b;
        public:
                void get(int x, int y)
                {
                        a=x;
                        b=y;
        }
Void display()
{
        cout<<"a="<<a;
        cout<<"b="<<b<<endl;
}
Friend Numbers operator * (Numbers n1, Numbers n2);
};
Numbers operator * (Numbers n1, Numbers n2)
{
                Numbers n;
                n.a=n1.a * n2.a;
                n.b=n1.b * n2.b;
                return n;
}
Void main()
{
        Clrscr();
        Numbers n1;
        n1.get(5, 7);
n1.dipslay();
Numbers n2;
n2.get(2,3);
n2.display();
Numbers n3= n1 * n2;// multiplying 2 objects
Cout<<\n after multiplying n1 and n2:\n";
n3.display();
getch();
}
```

Rules for the operator overloading

Although, it looks simple to overload the operator but it has some restrictions & limitations also which are as follow

1. Only existing operators can be overloaded.
2. You are not allowed to create new operators.
3. The overloaded operator must have at least one operand of user defined data-type.
4. We can't change basic meaning of operator means that you can't overload + operator to perform – operation.
5. There are some operators which can't be overloaded
    1) Class member access operator (.)
    2) Scope resolution operator (::)
    3) Conditional operator
    4) Size of operator
    5) When we overload the binary operator it must have *Return*

Type conversions

When you write an expression containing variables of different data types, type conversion is necessary (whether implicit or explicit).

For example,

```
float a=12.34;
int b= a;
```

Here, the value of a is transferred to b, but the fractional part will be truncated as the variable b is of type integer.

This also applies to class objects. For example consider following statements:

```
Test t1, t2, t3;
t1 = t2+ t3;
```

This is also valid as the t1, t2, and t3 are objects of same class and provided that a proper operator overloading function is defined.

But what if in an expression, there are objects of different class or combination of an object and a basic data type variable?

This can be done by performing proper type conversion method which can be one of the following:

1. Basic type to class type conversion

2. Class type to basic type conversion

3. One class to another class conversion

1. Basic type to class type conversion:

To understand this situation, consider a class Test and following statements

```
Test t1;
Int a;
t1=a; //int to class type conversion
```

To accomplish this, you have to make a constructor that takes an integer argument. Consider following example:

Example

```
#include<iostream.h>
#include<conio.h>
Class Time
{
        int hours, minutes;
        public:
```

```
        Time()
        {
                hours=minutes=0;
        }
        Time(int t)//constructor for int to class type conversion
        {
                hours=t/60; //getting hours from time
                minutes=t%60; //getting minutes
}
void display()
{

                cout<<"Hours:"<<hours<<endl;
                cout<<"Minutes:"<<minutes;
};
void main()
{
        clrscr();
        int t;
        cout<<"Enter time in minutes:";
        cin>>t;
        Time t1=t; //int to class conversion
        t1.display();
        getch();
}
```

***2. Class to basic type conversion**:*

In basic type to class type conversion, we created a constructor to perform the conversion but to perform class to basic type conversion, you have to define a conversion function for the type you want to convert, and the general form of the conversion function is:

```
operator basic_type_name()
{
        //conversion statements….
}
```

Here the basic_type_name can be any basic type data type such as int, float, double etc. you can define the operator function for the type want to convert into but the casting (conversion) operator function should meet following conditions:

1.The conversion function must be the member of class.

2.The function must not specify any return type.

3.It cannot take any argument.

Following example converts a class object to int type:

Example:

```
#include<iostream.h>
#include<conio.h>
class product
{
```

```
        int qty;
        float price;
public:
        Product(int q, float p)
        {
                qty= q;
                price=p;
        }
        void display()
        {
                cout<<"Quantity: "<<aty<<endl;
                cout<<"Price: "<<price<<endl;
        }
        Operator float() //conversion from class to float
        {
                return qty * price;
        }
};
void main()
{
        clrscr();
        Product p(10, 25);
        p.display();
        float amount;
        amount=p; //conversion from Product (class) to float
        cout<<"Total Amount: "<<amount;
        getch();
}
```

3. One class to another class conversion:

You may need to apply one class to another class conversion in some cases where in an expression there are objects of different classes. For example, consider following statements:

Obj1 = obj2;

Here obj1 and obj2 are objects of different classes.

We will implement this by simple method. In the following example, we will create two classes shop1 and shop2. We will create a constructor to implement conversion of one class object to another class object.

In the constructor we are going to aceess another class member variables. But generally for security reasons, the member variables are kept private. So to access those variables we will create functions to get values the member variables. This makes the member variables read-only as you can only read their values but you cannot modify them.

Example:

```
#include<iostream.h>
#include<conio.h>
Class Shop1
{
int code, qty;
```

```
float price;
public:
        Shop1(int c, int q, float p)
        {
                code = c;
                qty = q;
                price = p;
        }
        int getCode()
        {
                return  code;
        }
        int getQty()
        {
         Return qty;
        }
        float getPrice()
        {
                return price;
        }
        void display()
        {
                cout<<"Shop1 details……\n";
                cout<<"item code:"<<code<<endl;
                cout<<"quantity: "<<qty<<endl;
                cout<<"price: "<<price<<endl;
        }
};
Class Shop2
{
        int  code;
        float amount;
        public:
                Shop2(int c, float a)
                {
                        code = c;
                        amount = a;
                }
        void display()
        {
                cout<<"\n shop2 details….\n";
                cout<<"Item Code: "<<code<<endl;
                cout<<"Total Amount: "<<amount<<endl;
        }
        Shop2(Shop1 s1) //constructor to convert from one class to another
```

```
        {
                code= s1.getCode();
                amount = s1.getQty() * s1.getPrice();
        }
};
void main()
{
        clrscr();
        Shop1 s1(111, 5, 24.5);
        Shop2 s2 = s1; //shop1 to shop2 conversion
        s1.display();
        s2.display();
        getch();
}
```

Inheritance

Introduction:

→Reusability is the most important feature of OOP.

→It is always better because we can reuse something instead of creating the same again because it will save our time and memory as well as it will save us from fraction of as from error.

→This is our luck that C++ strongly supports the C++classes when developed it can be reuse in other classes.

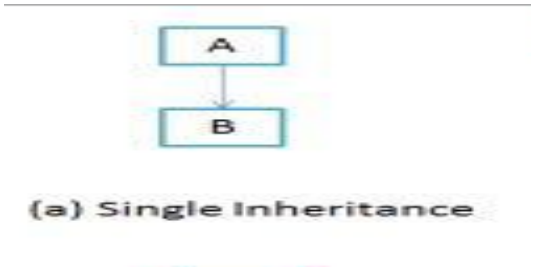
→This mechanism will work in which the old class is known as base class and the new class which have been developed on the base of the old class or sub as derived class here the whole mechanism is known as Inheritance .

➔There are five types of Inheritance

1) Single
2) Multiple
3) Hierarchical
4) Multi-level
5) Hybrid

1)Single Inheritance

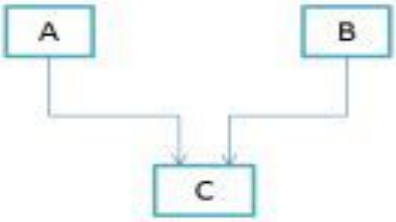
→When we declared the situation in which there is one base & one derived class is known as single inheritance.



(a) Single Inheritance

2) Multiple Inheritance

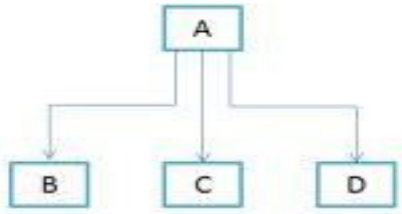
→In this type of inheritance more than one base class there is only one derived class is known as multiple inheritance.



(b) Multiple Inheritance

3)Hierarchical inheritance

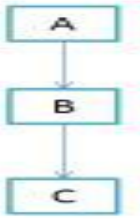
→In this type of inheritance there is one base class & more than one derived class.



(c) Hierarchical Inheritance

4) Multiple-level inheritance

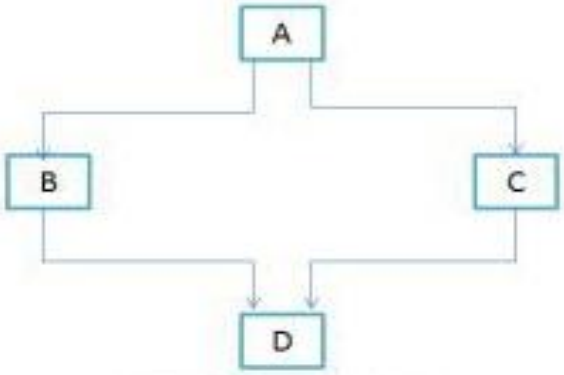
→In this type of inheritance has one base class one intermediate base class as derived class here it will represent the concept of grandfather, father & child.



(d) Multilevel Inheritance

5)Hybrid inheritance

→When we declare any two types of inheritance then it is represent hybrid inheritance.



(e) Hybrid Inheritance

How to declare inheritance

→When we want to have inheritance we need to declare at least 2 classes in which one class is base class (parent class) & another is derived class(child class).

→The syntax for the class declaration is as follows:

```
Class<derived class name> : <visibility mode><base name>
{
        --
        --
};
```

→In above syntax after the keyword class you have to write down the name of the derived class & than after the colon (:) sign, name of base class with its visibility mode which may be private or public by default is private.

→**Example:**

```
Class abc :        private xyz //private derivation
{
        --
        --
};
Class abc : public  xyz //public derivation
{
        --
        --
};
```

```
Class abc : xyz //private  derivation by default
{
        --
        --
};
```

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 the derived class. They are inaccessible to the object of the derived class. Remember that, a public member of a class can be accessed by its own objects using dot operator. The result is that no member of the base class is accessible to the objects of the derived class.

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 accessible to the object of the derived class. In both cases, the private members are not inherited and therefore, this private members of a base class will never become the members of its derived class.

Program:

```
#include<iostream.h>
#include<conio.h>
class b
{
        int a;
        public:
        int b;
        void get_ab();
        int get_a(void);
        void show_a(void);
};
class d:public b
{
        int c;
        public:
        void mul(void);
        void display(void);
};
void b::get_ab(void)
{
        a=5;
        b=10;
}
int b::get_a()
{
        return a;
}
```

```
void b::show_a()
{
        cout<<"a ="<<a<<endl;
}
void d::mul()
{
        c=b*get_a();
}
void d :: display()
{
        cout<<"a ="<<get_a()<<endl;
        cout<<"b ="<<b<<endl;
        cout<<"c ="<<c<<endl;
}
void main()
{
        d d1;
        clrscr();
        d1.get_ab();
        d1.mul();
        d1.show_a();
        d1.display();
        d1.b=20;
        d1.mul();
        d1.display();
        getch();
}
```

Method Overriding:

To override a method, a subclass of the class that originally declared the method must declare a method with same name, return type(or a subclass of that return type), and same parameter list.

The definition of the method overriding is:

- Must have same method name.
- Must have same data type.
- Must have same argument list.

Overriding a method means that replacing method functionality in child class. To imply overriding functionality we need parent and child classes. In the child class you define the same method signature as one defined in the parent class.

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
class base
{
    private:
            int  a, b;
    public:
```

```
        void display(void)
        {
                a=10;
                b=20;
                cout<<"A :"<<a <<endl;
                cout<<"b :"<<b<<endl<<endl;
        }
};
class derived: public base
{
    private:
        int x, y;
    public:
        void display(void)
        {
                x=50;
                y=100;
                cout<<"x  :"<<x<<endl;
                cout<<"y  :"<<y<<endl;
    }
};
void main()
{
    derived d;
    clrscr();
    d.display(); //call derived class function
    d.base::display();//call base class function
    getch();
}
```

Here first of all derived class function will be called and if we want to call base class function we have to use scope resolution operator with class name d.base::display().

Accessibility modes and Inheritance:

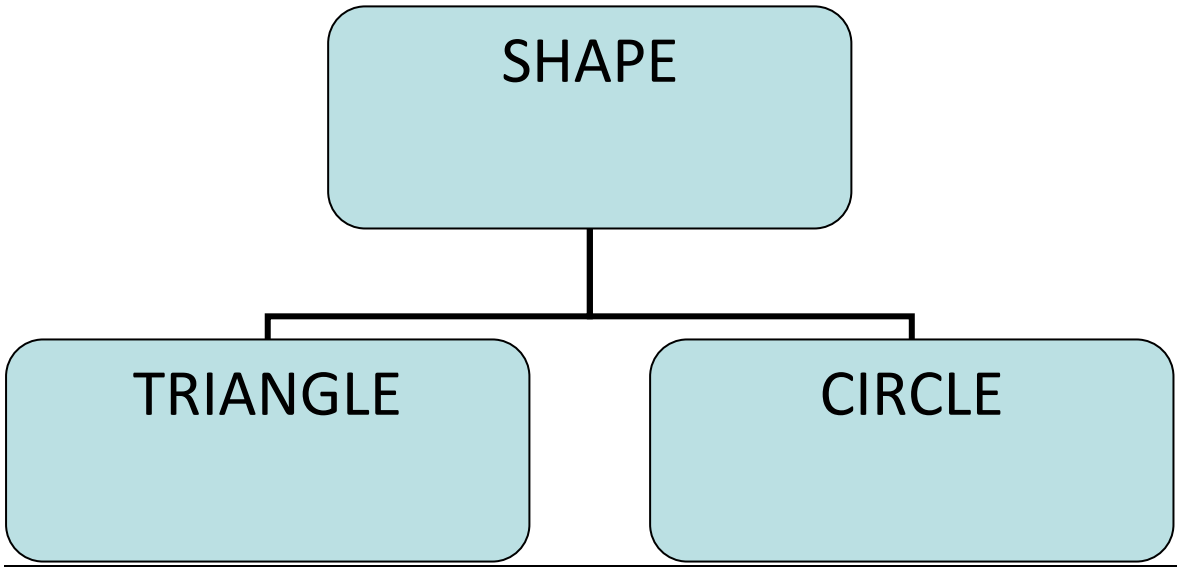
we can use the following chart for seeing the accessibility of the members in the Base class and derived class.

		Inheritance Mode		
		Public	Protected	Private
Members in base class	Public	Public	Protected	Private
	Protected	Protected	Protected	Private
	Private	X	X	X
		Member in derived Class		

Here X indicates that the members are not inherited, i.e. they are not accessible in the derived class.

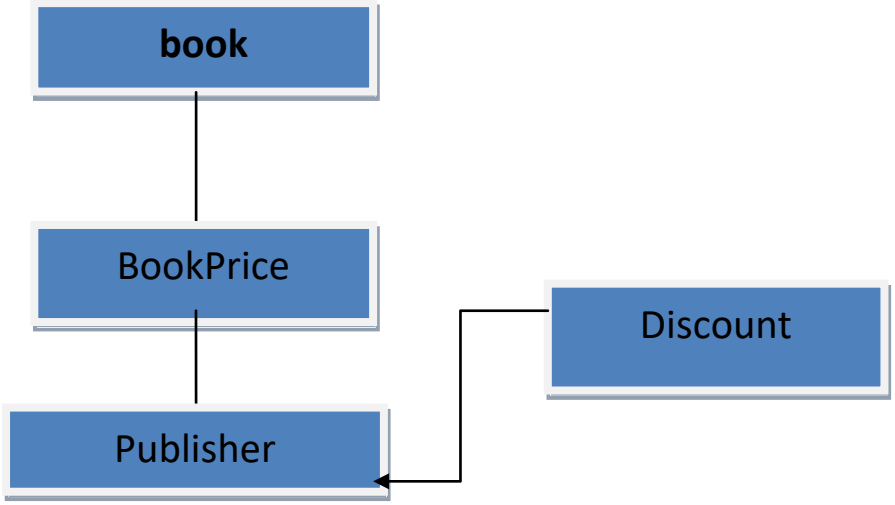
Hierarchical inheritance

In hierarchical inheritance, you can create hierarchical classification. You can create multiple classes from a base class. The new classes will acquire the properties of the base class and in addition they can have their own properties in the new classes. Following figure shows an example of hierarchical inheritance.



Hybrid inheritance

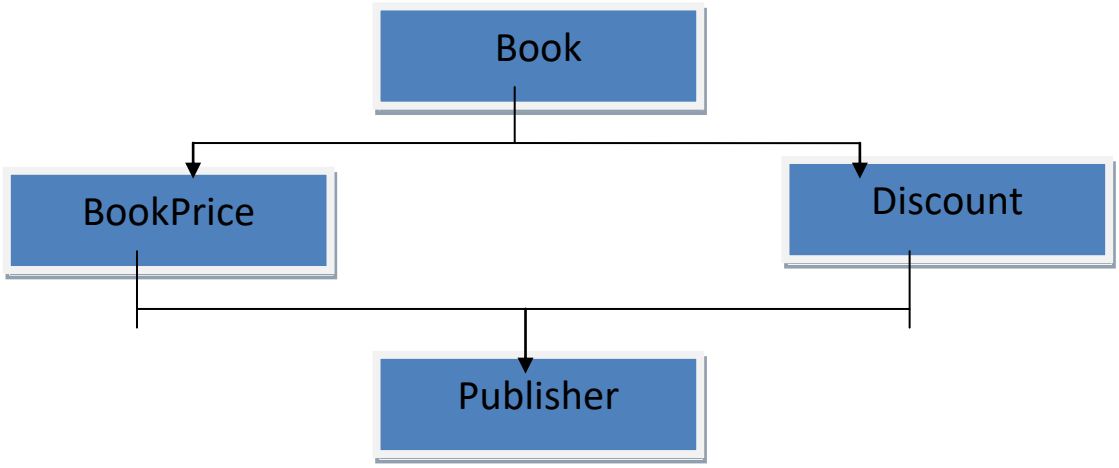
In hybrid inheritance, you can combine two or more types of inheritance. For example in the following figure we have combined two different types of inheritance namely multilevel and multiple inheritance.



Virtual base class

→In the above exp. we have used multiple & multilevel inheritance but
Some times we need that we have to use multiple, multilevel, hierarchical & Single inheritance in just one program,
→Consider the situation where we have a based class called grandfather from which we are preparing 2 derived class father &mother &from that also we will derive 1 child class this situation can be summarized in following figure,

→**Figure**



→Here in the above case if we consider the class of father &mother then it has the direct based class that is the grand father on the same way of the class child,

→We have 2 based classes will face one major problem that is a child class will retrieve public & protected member of grand father twice one from father &one from mother,

→In this situation to remove this C++ has given a new key-word called virtual

→This allows as transferring only one copy of data to the final derived class.

→**Note:-**

When you define the class as virtual &then you can write the key-word virtual can write virtual public class name& public virtual class name,

Abstract class:

→When we declare any class &if we are sure that no object will be defined for that class &that class is generally develop for providing base to other classes than it is know as abstract classes then it is know as abstract class instead of based class,

Constructor in derived class

In case of inheritance, if your base class contains a constructor with no arguments, the derived class does not need a constructor. But if the base contains a constructor with arguments then the derived class must have a constructor with argument.

If both the base class and derived class have constructors, the base class constructor is executed first and then the derived class constructor is executed. In multiple inheritance, the constructor are called in the order of the base class written. For example,

Class Derived: public Base1, public Base2

{

};

Here, the constructor of Base1 is called first as it appears first. In case of following example,

Class Derived : public Base2, public Base1

{

};

The constructor of Base 2 is called first and then the constructor of Base1.

In case of virtual base class, the virtual class constructor is called first. For **example ,**

Class Derived : public Base1, virtual public Base2

{

```
};
```

Here the Base2 is declared virtual so its constructor will be executed first.

In multilevel inheritance, the constructor are executed in order of inheritance i.e. the grandparent class constructor first, then the parent class and finally the child class constructor is executed.

This is shown in the following example:

```
#include<iostream.h>
#include<conio.h>
class Grandparent
{
        public:
                Grandparent()
                {
                        cout<<"Grandparent class"<<endl;
                }
};
class Parent :public Grandparent
{
        public:
                Parent()
                {
                        cout<<"Parent class"<<endl;l
                }
};
class Child : public Parent
{
        public:
                Child()
                {
                        cout<<"child class";
                }
};
void main()
{
        clrscr();
        Child c;
        getch();
}
```

Output:

```
Grandparent class
Parent class
Child class
```

Unit 4

Pointer, Virtual functions and Polymorphism, RTTI Console I/O operations

Polymorphism

→Polymorphism is most important features of OOP.

→It simply means one name multiple forms.

→Through operator overloading & method overloading, we have already seen the concept of polymorphism.

→In method overloading the member functions are selected by matching the argument type & no. of argument.

→This information known to the compiler at the run time, so the compiler is able to select the appropriate function.

→This is called as early binding, static binding or compile time polymorphism.

→Whenever, we have the same name function in based class as well as inderived class than which function will be called that will be depend on the object of the class calling it.

→Here the compiler doesn't know which function will be called

→This concept knows as late binding or dynamic binding or runtime polymorphism.

➔Pointer

➔Pointer to array

➔Array to pointer

➔Pointer to structure

➔Array [pointer] to structure

This pointer

→C++ uses a unique keyword called THIS, to represent an object that invokes a member function.

→This is the pointer that will point to the object for which the function has been called as THIS POINTER.

→For e.g. When we write a max () function means that when we are calling the max () function through the object A.

→Here the C++ mechanism will pass This pointer to the function & This pointer will points to the object A.

→Suppose, we have a situation in which we are calling one function with one object as argument than the object through which we are calling the function than This pointer will be set to point the calling object.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person
{
        private:
                char name[20];
                float age;
        public:
                person(char *s,float a)
                {
                        strcpy(name,s);
                        age=a;
                }
```

```
                person &person::greater(person &x)
                {
                        if(x.age>=age)
                        {
                                return x;
                        }
                        else
                        {
                                return *this;
                        }
                }
                void display(void)
                {
                cout<<"name="<<name<<endl;
                cout<<"age="<<age<<endl;
                }
};
void main()
{
        clrscr();
        person p1("Jonh",37.5);
        person p2("Hebber",20.0);
        person p3("Ivan",40.25);
        person p=p1.greater(p3);
        cout<<"Elder person is=";
        p.display();
        getch();
}
```

Pointer to derived class

→In C++, we have the most important concept is of inheritance on the other hand pointers are also most useful.

→When we have a based class as well as derived class, we can declare a pointer to the derived class through that pointer, we are able to call a member of based class.

Program:

```
#include<iostream.h>
#include<conio.h>
class base
{
        public:
                int b;
                void show(void)
                {
                        cout<<"b="<<b<<endl;
                }
};
class derived:public base
{
        public:
                int d;
```

```
                void show(void)
                {
                        cout<<"b="<<b<<endl;
                        cout<<"d="<<d<<endl;
                }
};
void main()
{
        clrscr();
        base *bptr;
        base bc;
        bptr=&bc;
        bptr->b=100;
        cout<<"bptr points to the base class"<<endl;
        bptr->show();
        derived dc;
        bptr=&dc;
        bptr->b=200;
//      bptr->d=300;
        cout<<"bptr points to the derived object";
        bptr->show();
        derived *dptr;
        dptr=&dc;
        dptr->d=300;
        cout<<"derived pointer to derived class";
        dptr->show();
        getch();
}
```

Virtual functions

→In polymorphism, when we have based class as well as a derived class & if we have a function with the same name with base class as well as derived class & if we have created one pointer object of based class.

→Now we assign the pointer object as the address of based class then defiantly it will call the based class members but when we define the pointer of based class & if it points to the derived class than also it will call the members of the based class.

→This will create a problem because without creating a derived class pointer we will not able to call the members of the derived class.

→If this is not possible than the concept of polymorphism will be violated so, C++ has given a special mechanism called Virtual function.

➔*Program*

```
#include<iostream.h>
#include<conio.h>
class base
{
        public:
                void display(void)
                {
                        cout<<"Display of Base class"<<endl;
                }
```

```
                virtual void show(void)
                {
                        cout<<"Shoe of Base"<<endl;
                }
};
class derived : public base
{
        public:
                void display(void)
                {
                        cout<<"Display of the Derived"<<endl;
                }
                void show()
                {
                        cout<<"Show of derived"<<endl;
                }
};
void main()
{
        clrscr();
        base b;
        derived d;
        base *bptr;
        derived *dptr;
        cout<<"Bptr points to the base"<<endl;
        bptr=&b;
        bptr->display();
        bptr->show();
        cout<<"dptr is points to the derived"<<endl;
        dptr=&d;
        dptr->display();
        dptr->show();
        getch();
}
```

Rules of virtual function

→When we declare any function as virtual, it must follow the following rules:

1) The virtual function must member of some class.
2) They can't be static.
3) They are accessed using pointer object only.
4) They must be with the same name with based class & derived class.
5) Constructor or destructor must not be Virtual.
6) If the virtual function is defined in the based & in the derived class than it must be invoked through the pointers.

Pure virtual function

→In normal practice, we have the virtual function in the same function will be in the derived class.

→Now when we have more than one derived class with same function than based class function will be called as Pure virtual function.

Run Time Type Identification (RTTI)

The Runtime type identification (RTTI) as its name suggests, lets you determine the type of an object at runtime.

The RTTI also allows you to check whether an object is of a particular type or whether two objects are of same type. This can be done by using typeid operator, which determines the actual type of its argument.

In c++, there are three main elements used for run-time type identification:

1.The **dynamic_cast** operator.

It is used for conversion between base class and derived class objects.

2.The **typeid** operator.

It is used for identifying the actual type of an object.

3.The **type_info** class.

This class stores the type information returned by the typeid operator.

CONSOLE I/O OPERATIONS

Introduction

→ In computer each & every program that we develop will take some data as I/p & generate process data as O/p.

→ In short, it will follow I/p-Process-O/p cycle.

→It is important to programmer to know hoe the data will be inputted & how the language gives the O/p.

→ In C++ we have used COUT & CIN for O/p & I/p respectively.

→C++ supports various functions &operators to control the O/p.

→ With this operators & function, we can get our desired O/p.

→For this purpose C++ use the concept of streams & stream classes.

C++ streams

A system is a sequence of Bytes.

→This acts as a source when we are inputting the data & it becomes the destination when the process to data has been O/p.

→The source stream that provides the data to the program is called as I/p stream & the destination stream that receives the data from the program is called as O/p stream.

→The data in I/p stream will came from the f\key-board or any other I/p device similarly the data in the O/p stream will be transferred to the stream or any other O/p device.

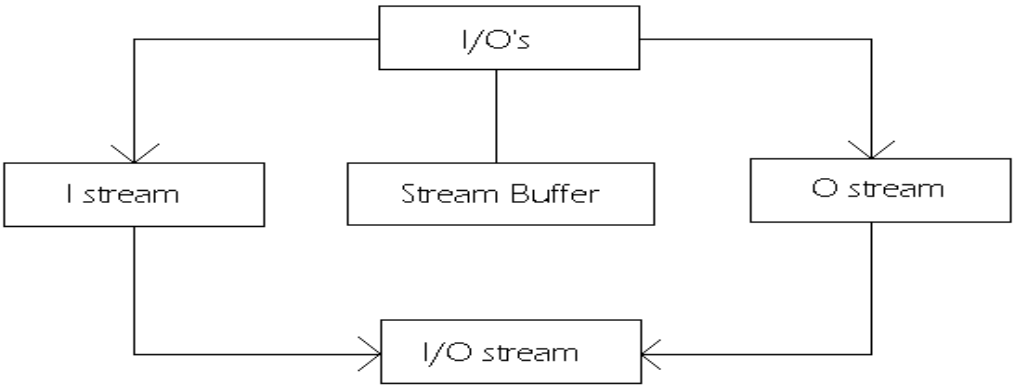
→C++contains several predefined streams that are automatically open when the programmer begins its execution at that moment C++also start the services like COUT & an in which monitor & key board are the default devices for I/p &O/p.

C++ stream classes

→ The C++ I/O system contains lots of classes that are useful for to define various streams to deal with consol as call as Disk File.

→The groups of class are called as stream classes & these classes are given in <IOSTREAM.H> header file which must be included in our program to communicate with consol.

→**Figure:**



→In above figure I/O's class is the base of class of I/p stream & O/p stream as well as from buffer stream.

→The Istream & Ostream are one base class of IOstream.

→I/O's class give the supports for formatted or unformatted O/p.

Unformatted I/O operation

→We have used the objects CIN & COUT for I/p & O/p of data.

→ This 2 objects have overloaded the operators >>&<< for I/p &O/p of basic data types.

→We already know the use of this objects & operators.

Put & Get function

→Put & Get functions are defined in Ostream & Istream classes respectively.

→Here the Get function will take any character has argument & put function will display, the given character on the screen.

Program of Getline function

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int count=0;
        char c;
        cout<<"Input any text=>";
        cin.get(c);
        while(c!='\n')
        {
                cout.put(c);
                count++;
                cin.get(c);
        }
        cout<<endl<<"No of character entered is==>"<<count;
        getch();
}
```

Program of Write function

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
```

```
        clrscr();
        char *s1="CPP";
        char *s2="Programming";
        int m=strlen(s1);
        int n=strlen(s2);
        for(int i=1;i<n;i++)
        {
                cout.write(s2,i);
                cout<<"\n";
        }
        for(i=10;i>0;i--)
        {
                cout.write(s2,i);
                cout<<"\n";
        }
        cout.write(s1,m).write(s2,n);
        cout<<"\n";
        getch();
}
```

Formatted I/O operations

→C++ supports various features that allow us to get the formatted O/p.
→For this purpose C++ has mainly three features:

1) I/O's class factions & flags.
2) Manipulators
3) User defined O/p functions

→The I/O's class provides various inbuilt functions for that formatted O/p functions this functions are as follows:

Function		***Use***
1) Width ()	→	to specify the sixe of the display of O/p value.
2) Precision ()	→	to specify the no. of decimal places to be displayed in the decimal numbers.
3) Field ()	→	to specify the character that is useful to fill the unused portion of given character.
4) Setf ()	→	to set the formatting flags to the control of O/p.
5) Unsetf ()	→	to clear the formatted flag set with Setf ().

→Just like I/O functions manipulator is also special kind of the formatted O/p.
→To use this manipulator, we must have to #include<iomanip.h>in our program.
→The manipulator & their equivalent functions are as follows

Manipulators		***Function***
1) Setw ()	→	Width()
2) Setprecision ()	→	Precision()
3) Setfield ()	→	Field()
4) SetIOsflags ()	→	Setw()
5) ResetIOsflags ()	→	Unset()

→In addition to this manipulator C++ also provides various other functions through which we can create our own manipulator also.

1) **Width ():-**

→ We can use this function to specify the width of our O/p.

→This is the member function of I/O's class so you can call it with help of COUT.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{
        clrscr();
        int item[4]={10,8,12,15};
        int cost[4]={75,100,60,90};
        cout.width(5);
        cout<<"Item"<<endl;
        cout.width(8);
        cout<<"Cost"<<endl;
        cout.width(15);
        cout<<"Total value"<<endl;
        int sum=0;
        for(int i=0;i<4;i++)
        {
                cout.width(5);
                cout<<item[i];
                cout.width(8);
                cout<<cost[i];
                int value=item[i]*cost[i];
                cout.width(15);
                cout<<value<<endl;
                sum=sum+value;
        }
        cout<<"grant Total="<<sum<<endl;
        getch();
}
```

2) **Precision ():-**

→By default the float value will print the six no's after the decimal place but we can also specify no. of digits to be displayed after the decimal place with the help of precision.

Program

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<iomanip.h>
void main()
{
        clrscr();
        cout.precision(3);
        cout.width(10);
        cout<<"Value"<<endl;
```

```
        cout.width(15);
        cout<<"Squre root of even value"<<endl;
        for(int i=1;i<5;i++)
        {
                cout.width(10);
                cout<<i;
                cout.width(15);
                cout<<sqrt(i)<<endl;
        }
        getch();
}
```

3)**Fill():-**

→When we want to print any value & if the even value has no. of characters will be less than it's given width at that the remaining places will be fill with white space.

→Instead of that, if we want to display any special character in blank space than we can use Fill function.

Program:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        cout.fill('<');
        cout.precision(3);
        for(int i=0;i<6;i++)
        {
                cout.width(3);
                cout<<i;
                cout.width(10);
                cout<<1.0/float(i);
                cout<<endl;
                if(i==3)
                {
                        cout.fill('>');
                }
        }
        cout<<"Padding change";
        cout.fill('#');
        cout.width(15);
        cout<<12.3456789<<endl;
        getch();
}
```

4)Setf():-

→When we use width function to print any value whether it is text or numeric, it will be always right justified.

→But in general techniquies, it is such that we will display left justified & no.'s right justified even when we printing no. in scientific notation at that time, it will be displayed left justified.

→ So, in all the situation, we have to use Setf () function.

→This function requires 2 arguments of I/O's class the list of format required & for that format which arguments are to be passed will be given in following table.

Format required	**Argument1**	**Argument2**
1)Left justifies	ios::left	ios::adjust field
2)Right justifies	ios::right	ios::adjust field
3)Padding with +/-	ios::internal	ios::adjust field
4)Scientific notation	ios::scientific	ios::float field
5)Fixed point notation	ios::fixed	ios::float field
6)Demal point notation	ios::dec	ios::base field
7)Octal point notation	ios::oct	ios::base field
8)hexadecimal point notation	ios::hex	ios::base field

→The above table requries 2 arguments, one flag & another is the Bit field.

→But if we want to display leading & trailing Zeros the setf function provides various other flags without Bit fields and take it as arguments this flags are as follows:

Flag	**Meaning**
ios::showbase →	Use the base indicate in the output.
ios::showpos→	Use the + sign before thpositive number.
ios::showpoint→	Display the tralling decimal points & zeros.
ios::uppercase→	For hexadecimal output the letters will be display in uppercase.
Ios::skipus→	Skip the white space in the output.
Ios::unitbuf→	Flush all the string after output.

Manging output with manipulators

→When we use formated output () of I/Os class then we have to use the object of the I/Os class that is cout.
→Instead of using these () we can use manipulators whixh are available in iomaip.h the list of manipualators are as follows.

	Manipulator	**Meaning**
1)	Setw (int w) →	Set the width to w.
2)	Setprecision (int d) →	Set the number of decimal palce to d
3)	Setfilled(char c) →	Set the filling char to c
4)	Setiosflags(long f) →	Set formated flag to f
5)	Resetiosflags(long f) →	Clear the formated flag
6)	Endl →	Insert a new line in the output.

→The main difference between ios functions and manipulators is when we want to apply the formated function to more then one output we have if we need to make formating in each and every output then we have to use the manipulators.

Unit 5

Working with Files, Exception handling, Introduction to Template STL

→In real life application when we develop any program data floppy disk or hard disk.

→A data stored on these devices will be informed of files.

→So that we can say that a file is a collection of related data stored in particular of the disk and our program perform read and write operator and it.

→When we have the concept of file handling to type of community:

1) Data transfer between console and program.
2) Data transfer between the program & file

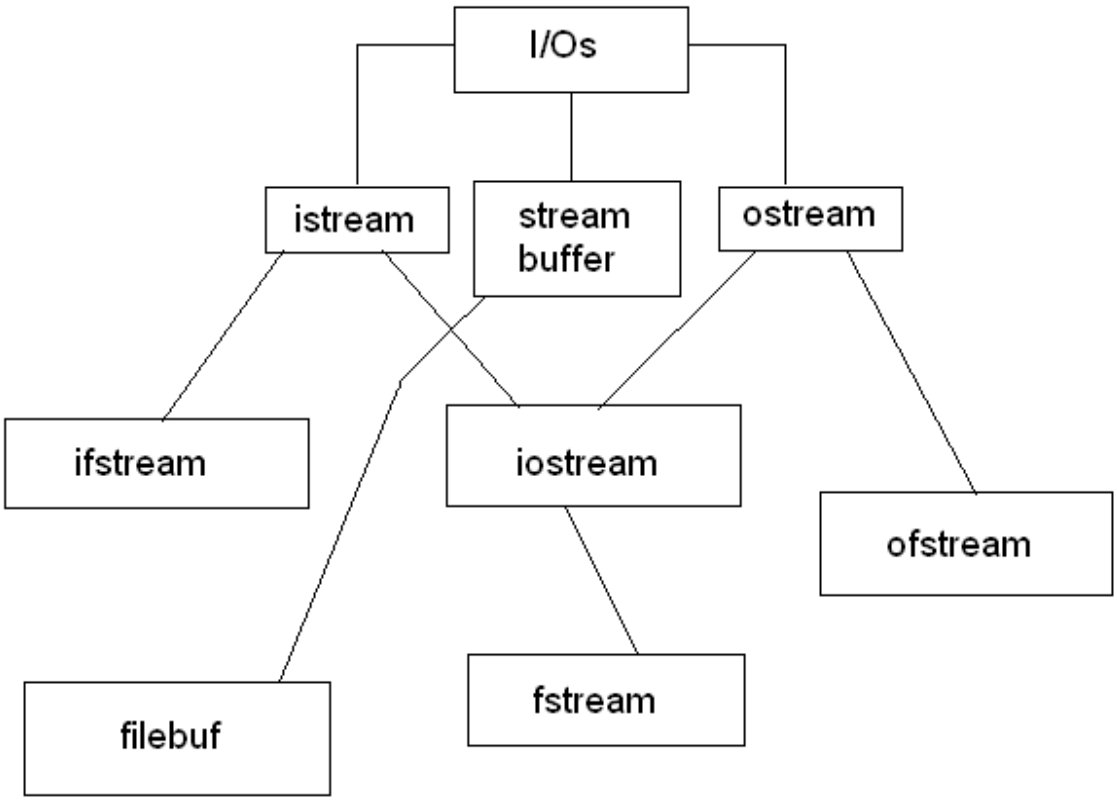
→In C++ file handling is very simple just like input/output operations in our routine programs.

→For reading the data from the file again the input stream will be useful for writing the data from the file to the console and the output stream will write the data from console to the file.

Classes for the file stream operations

→In C++ the IO system contain set of classes that defines file handling operations,

→These classes are ifstream, ofstream and fstream which can be summaries in the following figure:



Opening and closing of files

→When we are working with any files we have to decide following things

1) Suitable name for the file.
2) Data to be stored.
3) Purpose of the file.
4) File opening method.

→In C++, when we want to open the file we must create a file stream and than link it to the filename we can define file stream using ifstream, ofstream and fstream classes.

→These classes are available in fstream.h header file which class we have to use that we depend on there purpose that is we want read the data or we want to write the data whenever we want to open any file we have 2 ways.

1) Using a constructor of the class.
2) Using a function open () of the class.

→ The first method is very use when we use only one file in the stream which the second method is useful when we want to manage multiple files in a single stream.

Opening of file using constructor

→ We know that constructor is useful to initialize an object here we will use file name to initialize the file stream object.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        ofstream outf("item"); //connect item file to outf
        clrscr();
        cout<<"enter item name :";
        char name[30];
        cin>>name; //get name from keyboard and
        outf<<name<<endl; //write to file item
        cout<<"enter item coast :";
        float cost;
        cin>>cost; //get cost from key board and
        outf<<cost<<endl; //write to file item
        outf.close(); //disconnect item file from outf
        ifstream inf("item"); //connect item file to inf
        inf>>name; //read name from file item
        inf>>cost; //read cost from file item
        cout<<endl;
        cout<<"item name :"<<name<<endl;
        cout<<"item cost :"<<cost<<endl;
        inf.close(); //disconnect item from inf
        getch();
}
```

This program will create the file item.txt, and will put value of item name and item cost.

Ofstream outf(“item.txt”);

1) ofstream means “output file stream”. It creates a handle for a stream to write in a file.

2) outf –that’s the name of the handle. You can pick whatever you want.

3) (“item.txt”); - opens the file cpp-home.txt, which should be placed in the directory from where you execute the program. If such a file does not exist, it will be created.

Now, let's look a bit deeper. First, I'd like to mention that ofstream is a class. So, ofstream outf("item.txt") creates an object from this class. What we pass in the brackets, as parameter, is actually what we pass to the constructor. It is the name of the file. So, to summarize: we create an object from class ofstream, and we pass the name of the file we want to create, as an argument to the class constructor.

outf<<name<<endl<<cost<<endl; looks familiar like we have seen it in cout<<. This ("<<") is a predefined operator. Anyway, what this line makes is to put the text above in the file. As mentioned before, outf is a handle to the opened file stream. So, we write the handle name, << and after it we write the text in inverted commas. If we want to pass variables instead of text in inverted commas, just pass it as a regular use of the cout<<. This way:

outf<<variablename;

outf.close(); as we have opened the stream, when we finish using it, we have to close it. That is the close() function. So, we write the name of the handle, dot and close(), in order to close the file stream.

Notice: once you have closed the file, you can't access it anymore, until you open it again. We saw how to write into a file. Now, when we have item.txt, we will read it, and display it on the screen.

Reading A File

Ifstream inf("item.txt") – I suppose this seems a bit more familiar to you, already! ifstream means " input file stream". In the previous program, it was ofstream, which means "output file stream". The previous program is to write a file, that's why it was "output". But this program is to read from object from class ifstream, which will handle the input file stream. And in the inverted commas, is the name of the file to open.

Here inf>>name and inf>>cost

Read data from the file and store in variable and using cout print on screen. inf.close(); - as we have opened the file stream, we need to close it. Use the close() function, to close it. Note that once you have closed the file, you can't access it anymore, until you open it again.

OPENNING FILE USING THE MEMBER FUNCTION OPEN ():

As mentioned before, the above code creates an object from class ifstream, and passes the name of the file to be opened to its constructor. But in fact, there are several overloaded constructors, which can take more than one parameter. Also, there is function open() that can do the same job.

Here is an example of the above code, but using the open() function:

ifstream outf;

outf.open("item.txt");

what is difference here, just if you want to create a file handle, but don't want to specify the file name immediately, you can specify it later with the function open(). And by the way, other use of open() is for example if you open a file, then close it, and using the same file handle open another file. This way, you will need the open() function.

Use of open () with various file mode

→ Up till now, we have used open () with only one argument that is file name but this function can also take 2 argument that is file name and the file opening mode.

→Here the first argument the file name and second argument mode various file mode that we use as a second argument are as follows:

	File mode		Meaning
1)	ios::app	→	Append the data to the end of the file.
2)	ios::ate	→	Open the file and move the pointer to the end of the file.
3)	ios::in	→	Open the file for reading only.
4)	ios::nocreate	→	Open fail if the doesn't exist.
5)	ios::noreplace	→	Open fail is file already exists.

6) ios::out → Open the fail for writing only.

7) ios::trunc → Delete the content of the file if the data exist.

In fact, all these values are int constants from an enumerated type. But for making our life easier, we can use them as we see them in the table.

Here is an example on how to use the open modes:

```
#include<fstream.h>
void main()
{
        ofstream outf("file1.txt", ios ::ate);
        outf<<"that's  new! \n";
        outf.close();
}
```

As we see in the table, using ios::ate will write at the end of the file. If we didn't use it, the file will be overwritten, but as we use it, we just add text to it. So, running the above code, will add "that's new!" to it. You could create a file stream handle, which you can use to read/write file, in the same time! Here is how it works:

fstream outf("item.txt", ios ::in | ios:: out);

the code line above, creates a file stream handle, named "outf", as we know, this is an object from class fstream. When using fstream, we should specify ios::in and ios::out as open modes. This way, we can read from the file, and write in it, in the same time, without creating new file handles.

File pointer and their manipulators

→Each file has 2 pointers that are associated with the file they are known as file pointers.

→Out of 2 pointer is called as get pointer and another pointer is called is put pointer.

→The get pointer is useful to reading the data from the giver file location and the data from the give file location and the data to the given file location when we open in a read only mode then the get pointer will be always at the beginning of the file when wee open any file an append mode then put pointer will always at the end of file but if we open the file in a write mode then it will erase all the content of the file at the set the put pointer to the beginning of the file.

→To set the position according to our requirement we have been also given various other argument functions which will move the file pointer according to our argument.

→Functions are as follows:

Function		Meaning
1) seekg ()	→	Moves the get pointer to required location.
2) seekp ()	→	Moves the put pointer to the required place.
3) tellg ()	→	Gives the current position of the get pointer.
4) tellp ()	→	Gives the current position of the put pointer.

1) seekg():

The function seekg() is used with input streams, and it repositions the get pointer for the current stream to offset bytes away from origin or places the get pointer at position.

Syntax: seekg(offset, refposition):

Where 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 the 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

seekg function moves the associated file's get pointer. Below table list some sample pointer offset calls and their actions. fout is an ofstream object

For example, if you want to go 5 character back, you should write:

file.seekg(-5);

if you want to go 40 character after, just write:

file.seekg(40);

file.seekg(-5, ios :: end);

In this example, you will be able to read the last 4 characters of the text, because:
1)you go to the end (ios :: end)
2)you go to 5 characters before the end (-5)

Different argument with Seek ()

	Function		Meaning
1)	fout.seekg (o, ios::beg)	→	Go to the start position.
2)	fout.seekg (o, ios::cur)	→	Stay at the current position.
3)	fout.seekg (o, ios::end)	→	Moves the pointer to the end of the file.
4)	fout.seekg (m, ios::beg)	→	Moves the M bytes in the positive direction from beginning.
5)	fout.seekg (m, ios::cur)	→	Moves in the position direction for M bytes from the current position.
6)	fout.seekg (-m, ios::cur)	→	Moves the pointer in the backward direction from the current from the current position for M bytes.
7)	fout.seekg (-m, ios::end)	→	Moves in the backward direction for m bytes from the end of the file.

2)seekp():

Syntax: seekp(offset, refposition);

The seekp() function is used with output streams, but is otherwise very similar to seekg().

3)tellg():

Syntax: intvar tellg();

Returns the absolute position of the get pointer.

The get pointer determines the next location in the input sequence to be read by the next input operation. As integral value of type intvar with the number of characters between the beginning of the input sequence and the current position. Failure is indicated by returning a value of -1.

For example: len=inf.tellg(); where len is integer variable.

4)tellp():

The tellp() function is used with output streams, and returns the current put position of the pointer in the stream.

Syntax: **intvar tellg();**

Returns the absolute position of the put pointer. The put pointer determines the location in the output sequence where the next output operation is going to take place. An integral value of type intvar with the number of characters between the beginning of the output sequence and the current position. Failure is indicated by returning a value of -1.

For example cout<<"file pointer:"<<fout.tellp()

Sequential I/O operation

→There are two pairs of function available for performing I/O operation on file.

→One pair of function is put & get for handling single character and another is read & write to handle block data.

➔Put & get ()

→The put () useful to write a single character to the associated string while get () will read the single character from associated string.

For example file.put(char); , and file.get(char);

➔Read & Write ()

The read & write () will handle the data in the binary format as well as character format so whatever you write file stream that will be write an as it in the file.

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

outfile.read((char *) & v, sizeof(v));

these function takes two arguments. The first is the address of the variable v, and second is the length of that variable in byte. The address of the variable must be cast to type char * (i.e. pointer to character type).

➔Reading & Writing class object in the file

→Class & object are the central part of C++ so it puts natural that C++ has to allow the reading & writing of the class object in the file also for this purpose C++ has given read & write () which will be able to read & write class object also.

→Remember that with the help of read & write () you can write only the class data member & not the member functions.

Updating a file through random access

→ Updating is a general routine task the maintain of any data file updating including following things

1) Display the content of the file.
2) Add a new data.
3) Modify a data.
4) Delete a data.

→Whenever you want to perform any of the above operation you need to have the location where the updation will be done will be required to reach main that you have to reach to the location when the updation is needed.

Error handling during file operation

→Whenever face you work with any file we may encounter some errors in the following situation.

1) A file we attempt to open for reading does not exist.
2) A file name used for a new file may already exist.
3) We may attempt invalid operation like reading the data after end of the file
4) Their may not be space in the disk to store in the file.
5) We may use an invalid file name.
6) We may try to perform an operation for which the file is not open.

The ios class contains several useful error handling functions as listed below:

Function	Syntax	Description
eof()	int eof()	It returns a non-zero value if the pointer is reached at the end of file during read operation else returns zero.
fail()	int fail()	It returns a non-zero value when an input or output operation fails else returns zero.

good()	int good()	It returns a non-zero value if no error occurred during the last operation. Means the operation performed successfully.
bad()	int bad()	It returns a non-zero value if an error occurred during the last operation. Means an invalid operation was tried to perform.

Command line argument

Command line arguments are the arguments that are passed to the main function. These arguments are passed by the command at DOS prompt. So they are known as the command line arguments. As normal functions we cannot pass arguments to the main function as we don't call the main function explicitly but it is called implicitly when we run the program. So to pass the command line arguments, you have to run the program by the command and the arguments to the main function are passed as the supplementary commands.

Following is the syntax of the main function with command line arguments:

int main(int numofArgs, char *args[]);

Here, the first argument is number of arguments and second is array of arguments to be passed to the main function. The name of the program is passed as the first argument so it is also counted in the number of arguments.

The following is an example to demonstrate the command line arguments:

Example:

```
#include<iostream.h>
#include<conio.h>
void main(int n, char *a[])
{
        clrscr();
        cout<<"total arguments: "<<n<<endl;
        cout<<"Arguments are : "<<endl;
        for(int i=0; i<n; i++)
        {
                cout<<"Argument "<<i+1<<": <<a[i]<<endl;
        }
        getch();
}
```

EXCEPTION HANDLING

Exception handling

The each & every program that we develop may have errors when we run it first time there are mainly 3 types of errors

1) Compile time errors
2) Runtime time errors
3) Logical errors

1) Compile time errors

→Those type errors which generate you to poor understanding of the language and their rules.

→Generally the compiler have the capability of finding out most of the errors of this type if this type of errors is there than we are not able to run our program until we solve the errors.

2) Run time errors

→Runtime errors are those type of errors which are generated at the time of running the program or we can say at the time when we executed due to some wrong input from the user or sometime the user input is proper but due to some mathematical operation like division by o will also generate run time errors.

3) Logical error

→Logical errors are those types of errors which are generated due to the poor logic of the program in this type of error your program will be compile property run property but your output is not proper.

→Any of the three errors will give change to program to make misbehavior out of the three errors the solution of compile time error is comparatively easy and we have to make this errors to be solved because if we not solve this errors then our program will not run at all.

→For logical errors we do not have any solution but for runtime errors all the language provide some mechanism here in C++, we have been given mechanism of exception handling.

→**In exception handling, we have to perform following steps:**

1) Find the problem. (Hit the exception)
2) Inform that an error has been occurs.(Throw)
3) Receive the error information. (catch)
4) Take the corrective action. (Handle the exception)

→In C++, exception handling is built up on 3 keyword (Try, Throw and Catch).

→The try keyword is useful to cover the statement which generates the error when the exception has been generating it is by through statement and catch block catch the exception that is generated.

Program:

```
#include<iostream.h>
#include<conio.h>
void main ()
{
        int a, b;
        cout<<"Enter the A & B=";
        cin>>a>>b;
        int x=a-b;
        try
        {
                If (x! =0)
                {
                        cout<<"Result (a/x) ="<<a/x;
                }
                else
                Throw(x);
        }
        catch (int I )
        {
```

```
                cout <<"Exception generated";
            }
        getch ();
}
```

Introduction to Template STL

Template

→Template is one of the new features of C++.

→It is a concept while enables us to define generic classes and functions.

→So that, it provides support for generic programming generic program are approach where generic types are use in algorithm so that they work for verity of data types.

→A paragraph can be consider as a kind of macro here an object of a specific type is defined for actual use and then the template definition will be substituted by the specific data type.

→So the templates are sometimes called Parameterized classes.

→In templates the definition of class will be similar to ordinary class definition but here we will declare the one class object with the prefix template <class t> this prefix will tell the compiler that a template has been defined and it will use t as generic object.

Program:

```
#include<iostream.h>
#include<conio.h>
Const size=3;
Template <class t>;
Class vector
{
    T*v;
    Public:
        Vector ()
        {
            V=new T[size];
            For     (int I =0; i<size; i++)
            {
                V[i] =0;
            }
        }
        Vector (T*a)
        {
            For     (int I =0; i<size; i++)
            {
                V[i] = a[i];
            }
        }
        T operator * (vector &y)
        {
            T sum=0;
            For     (int I =0; i<size; i++)
            {
```

```
                                        Sum=sum+this→v[i]*y-v[i];
                                }
                                return sum;
                        }
        Void main ()
        {
                int x [3] = {1, 2, 3};
                int y [3] = {4, 5, 6};
                Vector<int>v1;
                Vector<int>v2;
                V1=x;
                V2=y;
                Int r=v1*v2;
                Cout<<"result="<<r;
                getch ();
        }
```

Class template with multiple parameters.

→When we declare any class template, we can also use more than one parameters which will be converted in more than one data type by using class template.

Program:

```
        #include<iostream.h>
        #include<conio.h>
        Class test
        {
                T1=a;
                T2=b;
                Public:
                Test (t1 x, t2 y)
                {
                        A=x;
                        B=y;
                }
                Void show ()
                {
                        cout<<a<<endl;
                        cout<<b<<endl;
                }
};
Void main ()
{
        Test<float, int>test1 (1.23, 123);
        Test<int, char>test2 (100, 'w');
        Test1.show ();
        Test1.show ();
        getch ();
```

```
}
```

Function template

→Like class template, we can also declare a function as template which will be useful to create family of function with different argument type.

→The syntax for the function template will be as follows::

```
Template<class t>|return type<function name>
        (Argument of type t)
{
        --
        --
};
```

Program:

```
#include<iostream.h>
#include<conio.h>
Template <class t>
Void main ()
{
        Fun (100, 200, 11.22, 33.44);
        getch ();
}
Void fun (int a, int b, float x, float y)
{
        Cout<<"A & B before swap"<<a<<b;
        Cout<<endl;
        Swap (a, b);
        Cout<<"A & B after swap"<<a<<b;
        Cout<<endl;
        Cout<<"X & Y before swap"<<x<<y;
        Cout<<endl;
        Swap (x, y);
        Cout<<"X & Y after swap"<<x<<y;
        Cout<<endl;
}
Void swap (t &m, t &n)
{
        T temp;
        Temp=m;
        M=n;
        N=temp;
}
```

overloading of template function

As the normal functions, a template function can also be overloaded. You can have same name of your function for template function and an ordinary function. This will result in function overloading. When a template function is overloaded, call to the function with same name will be resolved by first calling the

ordinary function having the exact prototype match. If it is not matched, then the template function is called.

If no function definition is matched with the function called, the compiler will generate an error.

Following program shows overloading of a template function:

Example:

```
#include<iostream.h>
Template <class Temp>
void  show(Temp a)
{
        cout<<"Template Function"<<endl;
        cout<<"value: "<<a<<endl<<endl;
}
void show(char a)
{
        cout<<"Normal function"<<endl;
        cout<<"value :"<<a<<endl<<endl;
}
void main()
{
        show(111);
        show("a");
        show(1.2);
}
```

Output:

```
Template Function
Value : 111
Normal Function
Value : a
Template Function
Value: 1.2
```

As we stated earlier, if an ordinary function is defined then the normal version of the overloaded function is called. So the normal function is called in case of char argument. In other cases, the template function is called.

Member function template

As we know that we can also have template functions as member functions of a class. The member function template can be defined inline as well as outside the class definition. In case of inline functions, the template functions are defined same as the functions we define for non-member functions. But to define the member function templates outside the class, you have to specify the type argument of the member function as show below:

```
Template <class Temp>
return_type <Temp> :: function_name(arguments)
{
        //function body
}
```

Following is the example of member function template:

Example:

```
#include<iostream.h>
Template <class Temp>
class Book
{
        int id;
        Temp price;
        public:
        void getdata(int i, Temp p);
        void display()
        {
                cout<<"Book ID: "<<id<<endl;
                cout<<"Book Price :"<<price<<endl;
        }
};
template<class Temp>
void Book<Temp> ::getdata(int i, Temp p)
{
        id=i;
        price =p;
}
void main()
{
        Book <int> b1;
        b1.getdata(111,210.10);
        b1.display();
        Book <double > b2;
        b2.getdata(222,211.15);
        b2.display();
}
```

Output:

Book ID:111
Book Price:210
Book ID:222
Book Price: 211.15

Introduction to STL (standard Template Library)

The standard Template Library is a set of general-purpose classes and functions that are used frequently by the programmers. These classes and functions are very useful that supports the idea of reusability and are used for storing and processing data that can be used for most programs.

The standard template Library contains a wide range of classes and functions with so many features. We shall discuss the important features of this library. Using STL you can save lot of time and code as it offers almost all the general-purpose functions that you may need in your programs.

To use the features of STL, you have to use the standard namespace by the following statement;

Using namespace std;

Overview of Iterators and Containers

The STL is comprised of following three main components:

-Containers
-Iterators
-Algorithms

The container can be considered as an object that contains the actual data. The containers are implemented by defining template classes so they support the concept of generic data types.

The Iterator is an object such as pointer that points to an element stored in a container. It is used to iterate (point to next element) when processing data elements of container such as searching, sorting, copying etc.

An algorithm is much like a function that processes the data stored in the container and it uses the iterator to iterate the elements of the containers. Algorithms are made of template function to support generic data types.

We will learn an example of vector container. A vector is a dynamic array which supports insertion and deletion of elements on it. You can create vector of any data type by specifying its type:

vector<int>v1;
vector<float>v2;

following are some useful functions defined in <vector>

size()- returns the number of elements contained in the vector.
capacity()- it returns the capacity of vector i.e.how many elements the vector can contain.
push_back()-Inserts the specified elements at the end of the vector.
Pop_back()-Removes the last element inserted.
clear()- Removes all the elements of the vector.

Example:

```
#include<iostream>
#include<vector>
using namespace std;
void display();
vector <int>v1;
vector<char>v2;
void main()
{
        cout<<"Initial Size: "<<v1.size()<<endl;
        cout<<"initial Capacity:"<<v1.capacity()<<endl;
        v1.push_back(111);
        v1.push_back(222);
        v1.push_back(333);
        v1.push_back(444);
        v1.push_back(555);
        cout<<"size after adding elements: "<<v1.size()<<endl;
        cout<<"Now Capacity: "<<v1.capacity()<<endl;
        display();
        cout<<"Deleting 3rd element"<<endl;
        v1.erase(v1.begin(),+2, v1.begin() +3);
        display();
        cout<<"deleting last element"<<endl;
        v1.pop_back();
        display();
```

```
        cout<<"deleting all elements"<<endl;
        v1.clear();
        display();
}
void display()
{
        cout<<"vector elements :[";
        for(int i=0; i<v1.size(); i++)
                cout<<v1[i]<<" ";
        cout<<"] "<<endl;
}
```

Output:

Initial Size :0
Initial Capapcity: 0
Size after adding elements : 5
Now Capacity: 6
Vector Elements: [111 222 333 444 555]
Deleting 3rd element
Vector Elements: [111 222 444 555]
Deleting last element
Vector Elements: [111 222 444]
Deleting all element
Vector Elements: [ ]