Language:-

A language is a source of communication.

A language is means for communication of any two persons or objects Can able to exchange information.

The computer languages are classified into following categories:-

1. Low level or assemble level
2. High level
3. Mid level

1. Low level language:-

This language is able to communicate with the system by using a Special instruction sets called "pseudo codes " or "symbolic codes"(Mnemonics) are used.

In order to translate this assembles language into machine language by using translator which is known as assembler.

Ex: - 8085, 8086programming

<u>Disadvantages of assembly language:-</u>

- The program written for one machine cannot be used for other machine; therefore knowledge of machine is required.
- The assembly language programming is tedious and time consuming.
- The programs consists of many instructions so program becomes large in size thus becomes difficult to maintain.

2. **<u>High level language</u>:-**

To overcome disadvantage of low-level language a high level language is developed which is similar to English language. It consists of simple English like statements.

Advantages:-

1. Need not to know the internal architecture to develop an app
2. It provides greater flexibility in learning, developing application

Similar to an assemble language the high level language also have a Translator. They are 2 types

1. Compiler
2. Interpreter

1. **Compiler**:-

This translator translates all the instructions of the program and writes the translated instruction as a file.

This file can be executed later. The source program is not need again until the program is changed.

Ex: - turbo C, C++

2. **Interpreter:-**

This translator translates each line of the program and then executes it immediately.

Since it does not store the translated version of the program as a file.

The source program is needed to execute the program again.

Ex: - Pascal, basics, java, shell programming (in UNIX)

Midlevel language:-

To overcome the disadvantage of low level

, the midlevel language is developed.

These are the most power full language and provide all kinds of flexibility during program development.

Ex:-C.

PROCEDURE -ORIENTED PROGRAMMING

Convention programming, using high level language such as COBOL, FORTRAN, and C, is commonly known as procedure-oriented programming.

In this approach data move openly around the system from function to function. Functions transform from one form to another.

Draw backs of pop:-

- Data move freely around the program and therefore vulnerable to change caused by any function in the program. It does not model very well the real world problem.

Disadvantages of C Language

1. C does not have OOPS feature that's why C++ is developed. If you know any other modern programming language then you already know its disadvantages.
2. There is no runtime checking in C language.

3. There is no strict type checking (for example**:** we can pass an integer value for the floating data type).
4. C doesn't have the concept of namespace.
5. C doesn't have the concept of constructors and destructors.

C&C++ differences:-

C

C++

1) C is a mid level language.

1) C++ is high level language.

2) It is procedure oriented programming language. 2) C++ OOPs programming language.

3) C program compile the top to bottom approach.   3) C++ program compile bottom to top

4) C language variables are declared first line of the   4) C++ variables are declared anywhere in the program.

5) C language Provides 32 keywords.
5) C++ provides 48 keywords.

6) C Provides 256 characters
6) C++ Provides 256 characters

7) C function can't provide overloading mechanism.   7) C++ Provides overloading mechanism.

8) C language at the time of function declaration   8) C++ supports default arguments.

Declaration it supports

It can't accept initialization of Parameters.

(Default argument)

9) C structures can't provide access specifier.

9) C++ supports access specifier.

(private, public, protected).

10) We can use functions inside structures in C++ but not in C.

In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

11). The NAMESPACE feature in C++ is absent in case of C

C++ uses NAMESPACE which avoid name collisions. For instance, two students enrolled in the same university cannot have the same roll number while two students in different universities might have the same roll number. The universities are two different namespace & hence contain the same roll number (identifier) but the same university (one namespace) cannot have two students with the same roll number (identifier)

12) The standard input & output functions differ in the two languages

C uses scanf & printf while C++ uses cin>> & cout<< as their respective input & output functions

13) C++ allows the use of reference variables while C does not

Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming

<u>Characteristics Of C++</u>:-

Portability:

You can practically compile the same C++ code in almost any type of computer and operating system without making any changes. C++ is the most used and ported programming language in the world.

Brevity:

Code written in C++ is very short in comparison with other languages, since the use of special characters is preferred to key words, saving some effort to the programmer (and prolonging the life of our keyboards!).

C Compatibility:

C++ is backwards compatible with the C language. Any code written in C can easily be included in a C++ program without making any change.

Speed:

The resulting code from a C++ compilation is very efficient, due indeed to its duality as high-level and low-level language and to the reduced size of the language itself.

Reusability:

The concept of inheritance provides an important feature to the object-oriented language-reusability. A programmer can take an existing class and, without modifying it, and additional features and capabilities to it. This is done by deriving a new class from an existing class.

History of C++:

C++ is the extension of c. c++ is an object-oriented programming language. It was developed by bjarne stroustrup at AT&T bell laboratories.

This language can be implemented by using simula67 and C language that supports oop.

Since the class was a major addition to the original c language, stroustrup initially called "c with classes", later and it is renamed as c++.

C++ = C + OOPS CONCEPTS

OUTPUT OPERATOR:-

The cout object sends data to the standard output device

syntax: cout<<data;

data can be…

-> variables,constants, expressions and combinations of all the three

Note: cout object is a standard output device of ostream stream class

INPUT OPERATOR:-

Cin>>number1;

It causes the program to wait for the user to type in a number.

→The identifier cin is a predefined object that represents the standarad input stream in c++.

<<

- ✓ It is called as extraction operator
- ✓ when it is used repeatedly in cout statement is called as cascading of output operator

>>

- ✓ It is called as insertion operator
- ✓ When it is used repeatedly in cin statement is called as cascading of input operator

The following is the basic structure of a c++ program

INCLUDE HEADER FILES

CLASS DECLARATION

CLASS FUNCTION

MEMBER FUNCTION DEFINATION

MAIN FUNCTION PROGRAM

C++ supports file manipulation in the form of stream objects.

The stream objects cin and cout are used to deal with the standard input and output devices.

These objects are predefined in **iostream.h** header file.

There are three classes for file handling:

ifstream - for handling input files
ofstream - for handling output files

Cin, cout objects are which included header file is iostream.h header file.

Rules to follow while writing programs:

- The whole source code is written in lowercase because the compiler is case sensitive
- Every statement is terminated by with a semicolon (;)
- We must save the file with .CPP extension. CPP stands for Cplusplus

Tokens:-

The smallest individual amount of a program is known as tokens.

1) Keywords
2) Identifiers
3) Constants
4) Variables
5) Operators.

6) Datatypes

1)<u>**keywords:-**</u>

They are explicitly reserved identifiers and can’t be used as names for the program variables or other user defined program elements.

1)asm 2)auto 3)break 4)case 5) catch 6)char 7)class 8)const 9)continue 10)default 11) delete 12)do 13)double 14)else 15)enum 16) extem 17)float 18)for 19)friend 20) goto 21)if 22)inline 23)int 24)long 25)new 26) operator 27) private 28)protected 29) public 30)register 31)return 32)short 33)signed 34)sizeof 35)static 36)struct 37)switch 38)templats 39)this 40)throw

41)try 42)typedef 43)union 44) unsigned 45)virtual 46) void 47)volatile 48)while

2) <u>Identifiers and constants:-</u>

Identifier refers to the names of variables, functions, arrays, classes, etc…….. Created by the programmer.

Constants refer to the fixed values that donot change during the execution of a program.

<u>Note-1:-</u>

C++ requires a const to be initialized. ANSI C does not require an initializer, if none is given , it initialzes the const to 0.

<u>Type compatibility:-</u>

C++ is very strict with regard to type compatibility as compared to C.

Another notable difference is the way **char** constants are stored.In c they are stored as **ints** and therefore.

sizeof('x')

Is equivalent to

sizeof(int)   in C.

In C++, however, char is not promoted to the size of int and ttherefore.

sizeof('x')

equals

sizeof(char)

<u>Declaration of variables:-</u>

In C all variables must be declared before they are used in executable statements. C requires all the variables to be defined at the beginning of a scope.

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

Reference variable:-

A reference variable provides an alias (alternative name) for a previously defined variable.

Syntax:-

Datatype & reference –name = variable – name;

Ex:-

int total=100;

```
int &sum=total;

cout<<sum;

cout<<total;
```

<u>Note:-</u>

Both the variables refer to the same data object in the memory.

```
#include<stdio.h>

#include<conio.h>

void f(int);

void main()

{

 int a=90;

 clrscr();

 printf("a value is=%d",a);
 f(a);
 printf("\nafter function calling a
value is=%d",a);
 getch();
 }
 void f(int a)
 {
 a=a+10;
 }
```

OUTPUT:-

```
a value is= 90
after function calling a value is=90
```

C++:-

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

```
#include<conio.h>
void f(int &);
void main()
{
 int a=90;
 clrscr();
 cout<<"a value is="<<a;
 f(a);
 cout<<"\n after function calling a
value is="<<a;
 getch();
 }
 void f(int &a)
{
```

```
            a=a+10;
            }
```

OUTPUT:-

```
a  value is= 90
after function calling a value is=100
```

Operators in c++:-

C++ has a rich set of operators. All c operators are valid in C++ also. In addition++ introduces some new operators.

<< insertion operator

>> extraction operator

:: scope resolution operator

::* pointer to member declaration

->*,.* pointer to member operator

new- memory allocation operator

delete- memory destroyed the allocated memory.

<u>Example on scope resolution:-</u>

:: variable will always refer to the global variable.

```
#include<iostream.h>

#include<conio.h>

int   m=10;    //global    m

void    main()

{

int  m=20;    //m     redeclared   local  to
main()

{
```

```
int    k=m;
int   m=30;      // m declared  again  local to
inner
block.
cout<< ” we are in inner block”;
cout<<”k=”<<k<<”\n”;
cout<<”m=”<<m<<”\n”;
cout<<”::m =”<<m<<\n”;
}
cout<<”we are in outer block\n”;
cout<<”m=”<<m<<”\n”;
cout<<”::m=”<<m<<”\n”;
getch();
}
```

Datatypes:

Userdefined data types:

Structures, class, union, enumerations

Built-in data types:

Integral type-int, char

Floating-float, double

Void

Derived data types:

Arrays, function, pointer, reference

Manipulators:

- ✓ Manipulators are operators used with the extraction operator (<<) to modify or manipulate the way data is displayed.

- ✓ Most commonly used manipulators are:

1) endl
2) setw

endl:

- ✓ Transfers the active cursor position to the next line then “flushes the output buffer”.
- ✓ It is similar to ‘\n’.

<u>Ex:-</u>

```
int   a=1234;
int   b=12;
int   c=123;
cout<<a<<endl;
```

```
cout<<b<<endl;

cout<<c;
```

<u>output:-</u>

```
1234

12

123
```

<u>Setw:</u>

- ✓ It is an operator that can be inserted directly into i/o statements to give space.
- ✓ The output is aligned from right to left.
- ✓ Which included header file <iomanip.h>

Ex:

cout<<setw(10)<<“countries”<<setw(15)<<“population”;

<u>Print the output in following format:-</u>

```
1234
  12
 123
```

Here, the numbers are right-justified.This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified.The **setw** manipulator does this job. **(iomanip.h)**

<u>Ex:-</u>

cout<<setw(5)<<a;

Type casting:-

Explicit type casting:-

C notation

(datatype)expression

C++notation

datatype(expression)

Ex:-

```
int   a=5,b=2;

float  c;

c=(float)a/b;        →c notation

c=float(a)/b;        →c++notation
```

DEFAULT ARGUMENTS:-

- C++ allows us to call function without specifying all its arguments.
- In such cases, the function assigns a default values to the parameter, when one or more argument omitted in call.
- Default values are specified when the function is declared.
- The compiler checks the function prototype with the arguments in the function call to provide values to those arguments, which are omitted.
- The arguments specified in the function

call explicitly always override the default values specified in the function prototype.

- One important point only the trailing arguments can have default values. We must add default arguments from right to left.

/*Sum of integer numbers using function with default argument*/

```
#include<iostream.h>

#include<conio.h>

int sum(int x=1,int y=2);  //function with
default arrugument

void main()

{
```

```
int a,b;
clrscr();
cout<<endl<<"ENTER A, B VALUES:";
cin>>a>>b;
int c=sum(a,b);
cout<<endl<<"\t SUM:"<<c;
c=sum(a);
cout<<endl<<"\t SUM:"<<c;
c=sum(b);
cout<<endl<<"\t SUM:"<<c;
c=sum();
cout<<endl<<"\t SUM:"<<c;
getch();
```

```
}
int sum(int x,int y)
{
cout<<endl<<"\n VALUE1:"<<x<<"\t
VALUE2:"<<y;
return(x+y);
}
```

//simple interest using function with default arugument

```
#include<iostream.h>
#include<conio.h>
double interest(long int p=100,int n=2,float
r=0.02);
void main()
```

```
{
long int principle;
int no;
float rate;
double si;
clrscr();
cout<<endl<<"ENTER PRINCIPLE NO OF
MONTHS AND RATE:";
cin>>principle>>no>>rate;
si=interest(principle,no,rate);
cout<<endl<<"\t SIMPLE
INTEREST:"<<si;
si=interest(principle,no);
```

```
cout<<endl<<"\t SIMPLE
INTEREST:"<<si;
si=interest(principle);
cout<<endl<<"\t SIMPLE
INTEREST:"<<si;
si=interest();
cout<<endl<<"\t SIMPLE
INTEREST:"<<si;
getch();
}
double interest(long int p,int n,float r)
{
cout<<endl<<"PRINCIPLE:"<<p;
cout<<endl<<"NO:"<<n;
```

```
cout<<endl<<"RATE:"<<r;
return(p*n*r/100);
}
```

FUNCTION OVERLOADING:-

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function **polymorphism** in oop.

The function would perform different operations depending on the type and number of arguments lists in the function call.

// some of two floating numbers using function overloading concept.

```
#include<iostream.h>
#include<conio.h>
int hello(int m,int n);
float hello(float p,float q);
void main()
{
int a,b;
clrscr();
cout<<endl<<"ENTER A B
VALUES(integers only):";
cin>>a>>b;
int c=hello(a,b);
```

```
cout<<endl<<"INTEGER'S MUL:"<<c;
float x,y;
cout<<endl<<"ENTER X Y VALUES(float
values only):";
cin>>x>>y;
float z=hello(x,y);
cout<<endl<<"FLOAT VALUES
SUM:"<<z;
getch();
}
int hello(int m,int n)
{return(m*n);}
float hello(float p,float q)
{
return(p+q);
```

```
}
```

/*w.a.p to exchant (swapping) two integer numbers, two floating numbers and two charcters using function overloading concept(using function with call by reference */

```
#include<iostream.h>
#include<conio.h>
void swap(int &,int &);
void swap(float &,float &);
void swap(char &,char &);
void main()
{
int a,b;
clrscr();
```

```
cout<<endl<<"ENTER A B
VALUES(integers only):";
cin>>a>>b;
cout<<endl<<"BEFORE SWAPPING
A:"<<a<<"\t B:"<<b;
swap(a,b);
cout<<endl<<"AFTER SWAPPING
A:"<<a<<"\t B:"<<b;
float x,y;
cout<<endl<<"ENTER X Y VALUES(float
values only):";
cin>>x>>y;
cout<<endl<<"BEFORE SWAPPING
X:"<<x<<"\t Y:"<<y;
swap(x,y);
cout<<endl<<"AFTER SWAPPING
```

```
X:"<<x<<"\t Y:"<<y;
char ch1,ch2;
cout<<endl<<"ENTER TWO
CHARECTERS:";
cin>>ch1>>ch2;
cout<<endl<<"BEFORE SWAPPING
CH1:"<<ch1<<"\t CH2:"<<ch2;
swap(ch1,ch2);
cout<<endl<<"AFTER SWAPPING
CH1:"<<ch1<<"\t CH2:"<<ch2;
getch();
}
void swap(int &m,int &n)
{
int temp=m;
    m=n;
```

```
      n=temp;
}
void swap(float &p,float &q)
{
float temp=p;
      p=q;
      q=temp;
}
void swap(char &c1,char &c2)
{
char temp=c1;
     c1=c2;
     c2=temp;
}
```

/*W.A.P to find the biggest elements of integer array and floating point array using function overloading concept.*/

```
#include<iostream.h>
#include<conio.h>
int big(int x[],int y);
float big(float s[],float z);
void main()
{
int i,n,a[20];
clrscr();
cout<<endl<<"\n ENTER HOW MANY N
ELEMENTS(integers only):";
cin>>n;
for(i=0;i<n;i++)
{
```

```
cin>>a[i];
}
int bi=big(a,n);
cout<<endl<<"\n BIGGEST
ELEMENT:"<<bi;
float p,q,r[20];
cout<<endl<<"\n ENTER HOW MANY Q
ELEMENTS(float values only):";
cin>>q;
for(p=0;p<q;p++)
{
cin>>r[p];
}
float si=big(r,q);
cout<<endl<<"\n BIGGEST
ELEMENT:"<<si;
```

```
getch();
}
int big(int x[],int y)
{
int b=x[0];
for(int i=0;i<y;i++)
{
if(x[i]>b)
b=x[i];
}
return(b);
}
float big(float s[],float z)
{
float d=s[0];
```

```
for(float p=0;p<z;p++)
{
if(s[p]>d)
d=s[p];
}
return(d);
}
```

Inline functions:-

An inline function is a function that is expanded in line when it is invoked. These are used to eliminate the cost of calls to small functions.

An inline function is a function that is expanded in one line. When it is invoked, that is compiler replaces the function call with the corresponding function code.

The inline function are defined as follows

Syntax:

```
inline returntype functionname(arg1...argn)
{
Function body
}
```

Some of the situations where inline function may not work:

- For functions returning values, if a loop, a switch, or a goto exists.
- For functions not returning values, if a return statement exists.
- If function contain static variables.

If inline functions are recursive.

NOTE:

If function definition is too long or too complicated and compile the function as a normal function.

// W.A.P to find product and division of two integer numbers.

```
#include<iostream.h>
#include<conio.h>
inline int product(int,int);
inline double division(int,int);
void main()
{
int a,b;
clrscr();
cout<<endl<<"ENTER A B VALUES:";
cin>>a>>b;
cout<<endl<<"PRODUCT:"<<product(a,b);
cout<<endl<<"DIVISION:"<<division(a,b);
getch();
}
int product(int x,int y)
{
return(x*y);
}
```

```
double division(int p,int q)
{
return(p/q);
}
```

OOP concepts:-

These are the concepts used extensively in object oriented programming.

- objects
- classes
- Data abstraction and encapsulation
- Inheritance
- polymorphism
- Data hiding.
- Message passing

Objects: objects are the instances of classes. Objects are the run time entities the may represent person, places etc.

classes:-

Classes are single unit it contains the variable declaration and function declaration.

In c++ terminology, variables are called data members, functions are called member function, and both are called members of particular class.

Data encapsulation:-

The wrapping up of data and functions in to a single unit is known as encapsulation.

Data abstraction:-

It refers the act of representing

essential features with out including the back ground details or explanation.

Since the classes use the concept of data abstraction they are know as abstract data types.

Inheritance:-

It is a mechanism of one class object a quires the properties of objects of another class.

Polymorphism:-

It is a mechanism the ability to make more than one form polymorphism is divided into 2 types

1. Compile time polymorphism (function overloading)

2. Run time polymorphism (operator overloading)

Data hiding:-

Using this mechanism to prevent the data from un authorized access.

Message passing:-

Providing the communication in between objects is called Message passing.

Class: A class is an extension to the structure data type. A class can have both variables and function as members

Class declaration:-

```
        Class classname
        {
```

Access specifier:

Member variable declaration

Member function declaration

Access specifier:

Member variable declaration

Member function declaration

};

- The body of a class is enclosed with in braces and terminated by a ;(semicolon)
- class body contains both variables declaration and function, both are collectively know as class member
- Class members that have been declared as private can be accessed only with in the class.
- Class members that have been declared

as public can be accessed from out side of the class.

Once the class has been declared we can create variable of type by using the class name i.e. create instance (object) for the class.

Access specifier:

It specifies what type of access we are providing to the data items i.e. variables placed in a class and member functions.
In C++ we have 3 types of access specifiers. They are

- → public
- → private
- → protected (used in inheritance concept)

Public:

If any variable or member function declared under public section then those can be accessed throughout the program or the project.

Private:

If any variable or member function declared under private section can be accessed only by the member function of the same class which is placed under a public section.

Protected:

It is similar to private access specifier but actual functionality is reflected in inheritance.

NOTE: we not mention access specifier then by default it will be consider as private.

OBJECT:

- ✓ Object is an instance of a class
- ✓ It is used to access class members and member functions
- ✓ Note: Memory will be allocated when we create an object for a class.

CREATING AN OBJECT:

Syn: classname   objectname

Note: class name & object name both are user defined like a structure & structure variable names.

Write a c++ programme to access members using private.

```
Class     one
{
     int   x;
     int   y;
     public:
     int   z;
};
Void main(  )
{         one  obj;
```

```
          obj  .  x  =  10  ;   //  error ,  x is
private
          obj  .  z   = 20  ;
     }
```

<u>Defining member functions:</u>

Member functions can be defined in two paces.

1.outside the class definition.

2.inside the class definition.

1.outside the class definition:

Member functions that are declared inside a class hare to be defined separately outside the class.

Syntax:

```
  Return-type classname::function-
name(argument decleration)
{
   Function body
}
```

Ex:

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

```
Class one
{
    Int number;
    Float cost;
     Public:
                  Void getdata(int,float);
                   Void display();
};
Void one::getdata(int a,float b)
{
Number=a;
Cost=b;
}
Void one::display()
{
```

```
Cout<<"number is"<<number;
Cout<<"\ncost is"<<cost;
}
void main()
{
One  obj;
Clrscr();
obj.getdata(10,23.45);
obj.display();
 getch();
}
```

o/p:___________;

w.a.p. to print the given number in reverse order.

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

```
#include<conio.h>
class reverse
{
private:
int n,revno;
public:
void setvalue(int no)
{
n=no;
}
void reverse_number(void);
void show(void);
};
void reverse::reverse_number(void)
{
```

```
int temp=n;
revno=0;
while(temp!=0)
{
int rev=temp%10;
revno=revno*10+rev;
temp/=10;       //temp=temp/10
}
}
void reverse::show(void)
{
cout<<endl<<" N:"<<n;
cout<<endl<<"REVERSE NO:"<<revno;
}
void main()
```

```
{
int value;
clrscr();
cout<<endl<<"ENTER ANY NUMBER:";
cin>>value;
reverse r1;
r1.setvalue(value);
r1.reverse_number();
r1.show();
getch();
}
```

/*Write a c++ programme to find Largest of two numbers:*/

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

```
class set
{
int m,n;
public:
 void input();
 void display();
 int largest();
 };
 int set::largest()
 {
 if(m>=n)
 return m;
 else
 return n;
 }
```

```
void set::input()
{
cout<<"\n enter values of m and n"<<endl;
cin>>m>>n;
}
void set::display()
{
cout<<"m="<<m<<" "<<"n="<<n<<endl;
//cout<<"largest value="<<largest()<<endl;
}
void main()
{
clrscr();
set a;
a.input();
```

```
 a.largest();
 a.display();
 getch();
 }
```

2.inside the class definition:

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

Ex:

```
//……………………………………….header
files
declaration………………………………………
…….
#incude<iostream.h>
#include<conio.h>
```

```
//…………………………class
declaration………………………………
………….
Class one
{
   Int number;
   Float cost;
    Public:
                 Void getdata(int,float);
//declaration
                  Void display()
//definition inside the class
                  {
                   Cout<<"number is"<<number;
                   Cout<<"\ncost is"<<cost;
                  }
```

```
};
//………member function definition…………………………………
Void one::getdata(int a,float b)
{
Number=a;
Cost=b;
}
//……………………main program…………………………………………
void main()
{
One  k;
Clrscr();
k.getdata(11,23.456);
```

```
k.display();
 getch();
}
o/p:__________.
```

//w.a.p to find add,sub,mult,div,rem of two integer numbers

`

```
#include<iostream.h>
#include<conio.h>
class binary
{
private:
```

```
int a,b;  //datamembers
public:
void getvalues(void)
{
cout<<endl<<"ENTER A,B VALUES:";
cin>>a>>b;
}
void show(void)  //member function
{
cout<<endl<<"SUM   :"<<a+b;
cout<<endl<<"SUB   :"<<a-b;
cout<<endl<<"MUL   :"<<a*b;
cout<<endl<<"DIV   :"<<a/b;
cout<<endl<<"REM   :"<<a%b;
}
```

```
};
void main()
{
binary b1;
clrscr();
b1.getvalues();
b1.show();
getch();
}
```

Private member functions:

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

Ex:

```
 Class item
```

```
{
      Int no;
      Void read();                          //  private
member function
      Public:
              Void update();
               Void write();
};
If  s is an object of item,then
s.read();                                //won't work:
objects cannot access private members
void item::update()
{
      Read();                         //simple call; no
object used.
}
```

Nesting of member functions:-

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

Ex:-

```
#include<iostream.h>
#include<conio.h>
class  set
{
     int  m , n ;
     public :
        void input ( ) ;
        void display( );
        int    largest( );
};
int      set     : :   largets( )
{
     if ( m > = n )
     return m;
     else
     return n;
}
void   set   : :   input ( )
{
     cout   <<  "  enter  m , n values";
     cin     >>   m >>  n;
```

```
    }
    void  set   ::  display ( )
    {
        cout   <<   "  larget value  = "
<<largest ( ) << " \n";
    }
    void   main ( )
    {
             set       s;
        s  .  input ( ) ;
        s  .  display  (  ) ;
    }
```

//w.a.p to find the factorial of given number using nesting of member functions.

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

```
#include<conio.h>
class factorial
{
private:
int no,fact;
public:
void setvalue(int val)
{
no=val;
}
int factorial_number(void);
void display(void)
{
cout<<endl<<" NO:"<<no;
cout<<endl<<"FACTORIAL:"<<factorial_n
umber();
}
};
int factorial::factorial_number(void)
{
int f=1;
for(int i=1;i<=no;i++)
{
f*=i;  //f=f*i;
}
return(f);
}
void main()
{
```

```
factorial f1,f2,f3;
clrscr();
f1.setvalue(4);
f2.setvalue(5);
f3.setvalue(6);
f1.display();
f2.display();
f3.display();
getch();
}
```

Arrays with in a class:-

```
#include<iostream.h>
#include<conio.h>
const  int m=50;
class ITEMS
{
    int       itemCode[m];
    float    itemPrice[m];
    int       count;
    public:
    void         CNT()
    {
        count=0;
    }
    void        getItem();
    void        displaySum();
    void        remove();
    void        displayItems();
};
void    ITEMS  : :       getItem()
{
    cout<<"Enter item Code:";
    cin>>itemCode[count];
    cout<<"Enter item cost:";
    cin>>itemPrice[count];
    count++;
}
```

```
void      ITEMS    : :    displaySum()
{
    float sum=0;
    for(int i=0;i<count;i++)
    sum=sum+itemPrice[i];
    cout<<"\n Total value:"<<sum<<"\n";
}
void      ITEMS    : :     remove()
//delete a specified item
{
    int     a;
    cout<<"Enter item Code:";
    cin>>a;
    for(int i=0;i<count;i++)
    if(itemCode[i]==a)
    itemPrice[i]=0;
}
void ITEMS : : displayItems()
//displaying items
{
    cout<<"\n Code Price\n";
    for ( int  i=0 ; i < count ; i++ )
    {
        cout<<"\n"<<itemCode[i];
        cout<<"  "<<itemPrice[i];
    }
    cout<<"\n";
}
```

```
int main()
{
          ITEMS     order;
     order.CNT();
     int x;
     do
     {
          cout<<"\n you can do the
following:";
          cout<<"\n enter appropriate number
\n";
          cout<<"\n1:Add An Item";
          cout<<"\n2:Display Total Value";
          cout<<"\n3:Delete An Item";
```

```
        cout<<"\n4:Display All Items";
        cout<<"\n5:Quit";
        cout<<"\n \n what is ur option?";
        cin>>x;
        switch(x)
        {
            case 1:order.getItem(); break;
            case 2:order.displaySum(); break;
            case 3:order.remove(); break;
            case 4:order.displayItems();
break;
            case 5:break;
            default:cout<<"error in input:try
again \n";
```

```
            }


      } while(x!=5);
      return 0;
}
```

Arrays of objects:-

```
      #include<iostream.h>
      #include<conio.h>
      class employee
      {
        char name[30];
        int age;
        public:
```

```
  void getdata();
  void putdata();
};
void employee::getdata()
{
cout<<"enter name:";
cin>>name;
cout<<"enter age";
cin>>age;
}
void employee::putdata()
{
 cout<<"name =  "<<name<<"\n";
 cout<<"age =   "<<age<<"\n";
}
const int size=3;
void main()
{
  employee emp[size];
  clrscr();
  for(int i=0;i<size;i++)
  {
   cout<<"enter details of
employee"<<i+1<<"\n";
   emp[i].getdata();
   }
   for(i=0;i<size;i++)
```

```
{
 cout<<"\nemployee"<<i+1<<"\n";
 emp[i].putdata();
 }
 getch();
 }
```

Objects as arguments:-

Like any other data type , an object may be used as a function argument. This can be done in two ways.

- ➔ A copy of the entire object is passed to the function.
- ➔ Only the address of the object is transferred to the function.

The first method is called pass-by-value . The second method is called pass-by-reference.

Objects as arguments:-

```
 #include<iostream.h>
#include<conio.h>
class time
{
 int hours,minutes;
 public:
 void gettime(int h,int m)
 {
 hours=h;
 minutes=m;
 }
 void puttime()
 {
 cout<<hours<<" hours and";
 cout<<minutes<<" minutes"<<"\n";
 }
 void sum(time,time);//declaration with
objects as arguments
};
void time::sum(time t1,time t2)
{
minutes=t1.minutes+t2.minutes;
hours=minutes/60;
minutes=minutes%60;
```

```
hours=hours+t1.hours+t2.hours;
}
void main()
{
time t1,t2,t3;
t1.gettime(2,45);     //get t1
t2.gettime(3,30);     //get t2
t3.sum(t1,t2);
cout<<"t1=";t1.puttime();     //display  t1
cout<<"t2=";t2.puttime();    //display   t2
cout<<"t3=";t3.puttime();    //display   t3
getch();
}
```

FRIEND FUNCTION:

Where we would like two classes to share a particular function.

In such situations C++ allows the common function to be made friendly with both the classes, there be allowing the function to have access to private data of these classes.

Such function need not be a member of any of these classes.

friend declaration:

```
class sample
{
```

```
------------

------------

Public:

friend returntype
functionname(arg1.....argn);

};
```

A Friend function has special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- It cannot be called using the object of that class.
- It can be declared either in the public or private part of the class without affecting its meaning.
- It has the objects as arguments.

- Friend functions are often used in operator overloading.

//w.a.p to access the private data of two classes and find biggest elements

Ex:-

```
#include<iostream.h>

#include<conio.h>

class second;

class one

{

 int x;

 public:

 void setvalue(int i)

 {

  x=i;
  }
  friend void max(one,second);
};
class second
{
 int a;
 public:
 void setvalue(int i)
 {
  a=i;
 }
 friend void max(one,second);
};
```

```
void max(one m,second n)
{
  if(m.x>=n.a)
    cout<<"\n big number is="<<m.x;
  else
    cout<<"big number is="<<n.a;
}
void main()
{
int a,b;
  clrscr();
  cout<<"enter any two numbers";
  cin>>a>>b;
```

```
 one o;
 second s;
 o.setvalue(a);
 s.setvalue(b);
 max(o,s);
 getch();
 }
```

The function max has arguments from both <u>**one**</u> and <u>**second**</u>. When the function max() is declared as a friend in **one** for the first time, the compiler will not acknowledge the presence of **second** unless its name is declared in the beginning as

class second ; → this is known as forward declaration.

Note:-

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator.

```
class   x
{
      …………….;
      ……………..;
            int   fun1();            //   member
function of x
      ………………;
};
class   y
```

```
{
      ………….;
      ………….;
      friend  int   x  : :  fun1();   //  fun1() of   x
is friend of    y
      ………….;
};
```

<u>Ex:-</u>

```
#include<iostream.h>
#include<conio.h>
class one
{
  void show();
};
class two
```

```
{
  friend void one::show();
};
void show()
{
cout<<"show function";
}
void main()
{
 clrscr();
 show();
 getch();
}
```

<u>**Returning objects:-**</u>

A function cannot only receive objects as arguments but also can return them.

Ex:-

```
#include<iostream.h>
#include<conio.h>
class complex
{
 float x,y;
 public:
 void input(float a,float b)
 {
 x=a;
 y=b;
 }
 friend complex sum(complex,complex);
 void show(complex);
};
complex sum(complex c1,complex c2)
{
 complex c3;
 c3.x=c1.x+c2.x;
 c3.y=c1.y+c2.y;
return c3;
}
void complex:: show(complex c)
{
cout<<c.x<<"    "<<c.y<<"\n";
```

```
}
void main()
{
complex a,b,c;
clrscr();
a.input(3.10,5.65);
b.input(2.75,1.20);
c=sum(a,b);
cout<<" a = ";
a.show(a);
cout<<" b = ";
b.show(b);
cout<<" c = ";
```

```
 c.show(c);
 getch();
 }
```

w.a.p to exchange the private data of two class using friend function

(function using call by reference)*/

```
#include<iostream.h>
#include<conio.h>
class second;   // forward declaration
class first
{
private:
int x;
public:
void getx(void)
```

```
{
cout<<endl<<"ENTER X VALUE:";
cin>>x;
}
void showx(void)
{
cout<<endl<<"\t X:"<<x;
}
friend void swapping(first &f,second &s);
//friend function
};
class second
{
private:
int y;
```

```
public:
void gety(void)
{
cout<<endl<<"ENTER Y VALUES:";
cin>>y;
}
void showy(void)
{
cout<<endl<<"\t Y:"<<y;
}
friend void swapping(first &f,second &s);
//friend function
};
void swapping(first &f,second &s)
{
```

```
int temp=f.x;
     f.x=s.y;
     s.y=temp;
}
void main()
{
first f1;
second s1;
clrscr();
f1.getx();
s1.gety();
cout<<endl<<"BEFORE SWAPPING:";
f1.showx();
s1.showy();
swapping(f1,s1);
```

```
cout<<endl<<"\n AFTER SWAPPING:";
f1.showx();
s1.showy();
getch();
}
```

Static data members:

A data member of a class can be qualified as static. It is initialized to zero when the first object of its class is created. No other initialization is permitted.

Ex:

```
#include<iostream.h>
Class item
{
    Static int count;
    Public:
            Void display();
};
Void item::display()
{
Cout<<"count is"<<count;
}
Int item::count;                    //definition
of static data member.
Void main()
{
Item i;
i.display();
}
o/p:_____.
```

Write a c++ programme to display Static Data Members

```
#include<iostream.h>

#include<conio.h>

class sample

{

private:

int x;

static int count;   //static data member

public:

void setvalue(int p)

{

x=p;

count++;
}
void showvalue(void)
{
cout<<endl<<"\t X: "<<x;
}
void showcount(void)
{
cout<<endl<<"COUNT :"<<count;
}
};
int sample::count;     //defined static data
member
void main()
{
```

```
sample s1,s2,s3;
clrscr();
s1.showcount();
s2.showcount();
s3.showcount();
s1.setvalue(10);
s2.setvalue(20);
s3.setvalue(30);
s1.showvalue();
s2.showvalue();
s3.showvalue();
s1.showcount();
s2.showcount();
```

```
s3.showcount();
getch();
}
```

Static member functions:

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties.

1. A static function can have access to only other static members (functions or variables)declared in the same class.

2. A static member function can be called using the class-name.

Syntax:

```
        Class-name :: function -name ;
```

Ex:

```
#include<iostream.h>
#include<conio.h>
Class   sample
{       Int no;
        Static int count;                          //static
member variable
        Public:
                Void getdata();
                Static void putdata()
//static member function
```

```
        {
            Cout<<"count is:"<<count;
            Cout<<"no is:"<<no;
//error: no is not static
        }
};
Int sample::count;
Void sample::getdata()
{
No=10;
Count++;
}
Void main()
{
```

```
Sample    s;
s.getdata();
sample::putdata();
getch();
}
o/p:_________________.
```

//w.a.p to count the number of objects created in the following program

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

```
private:
int a,b;
static int count;    //static data member
public:
void setvalue(int x,int y)
{
a=x;
b=y;
count++;
}
void showvalue(void)
{
cout<<endl<<"\n\t A:"<<a<<"\t B:"<<b;
}
static void showcount(void)
//void showcount()
{
cout<<endl<<"TOTAL NO OF OBJECTS
CREATED:"<<count;
}
};
int sample::count;
void main()
{
sample A;
clrscr();
A.setvalue(10,20);
```

```
A.showvalue();
A.showcount();
//sample::showcount();
sample B,C;
B.setvalue(30,40);
B.showvalue();
B.showcount();
//sample::showcount();
C.setvalue(80,90);
C.showvalue();
C.showcount();
//sample::showcount();
getch();
```

Constructors:

- ✓ Constructor is a special member function which is used to initialize the object i.e. it assigns the default values for data items which are defined inside a class.
- ✓ If the variable is

1) Integral type then it assigns zero
2) char or string type it assigns '\0'
3) Pointer or reference type the it assigns "NULL".

Note: - empty class always takes 1 byte of memory

RULES FOR DECLARING CONSTRUCTORS:

- Constructor name should be same as class name(case sensitive)
- It should be declared under public section (if we define under private then we cannot create object)
- It doesn’t return any value, not even void, and they cannot return values.
- It is invoked automatically when we create an object.
- Constructors can be overloaded.

TYPES OF CONSTRUCTORS:

In C++ we have 3 types of constructors

1) Default constructor or empty constructor

2) Parameterized constructor
3) Copy constructor

Default constructor:

Constructor which does not take any arguments is known as default constructor.

- ✓ It is optional to specify definition of default constructor when we create normal object.
- ✓ When we are not providing default constructor C++ compiler supplies its own default constructor for initialization process

Syntax:

```
    class classname
   {
   private member
 public:
    classname();
  };
classname::classname()
{
////
}
```

Ex:

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

```
{
    Int no;
    Public:
    Abc();
//constructor declared
    Void display();
};
Abc::abc()                              //constructor
defined
{
No=10;
}
Void abc::display()
//member function defined
{
Cout<<"no is"<<no;
}
```

```
Void main()
{
Clrscr();
Abc c;
//by calling constructor implicitly
Abc c1=abc();
//by calling constructor explicitly
c.display();
c1.display();
Getch();
}
o/p:__no is 10
     no is 10____.
```

Parameterized constructor:

- Constructor which takes arguments or parameters is called as parameterized constructor.
- It is compulsory to specify definition for parameterized constructor.
- If may be necessary to initialize the data member of all objects to some value instead of zero.
- C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.

- So those constructors that can take arguments are called parameterized constructors.
- We must pass the initial values as arguments to constructor function when object is declared. This can be done in two ways.
- By calling the constructor explicitly
- By calling the constructor implicitly.

<u>Syntax:</u>

```
 class classname
{
 private member;
public:
 classname(args1…argsn);
};
```

```
  classname::classname(args1…argsn)
  {
  ////
  }
```

Ex:

```
#include<conio.h>
Class program
{
  Int m,n;
   Public:
    program (int,int);
//constructor declaration(or)prototype
       Void display();
//member function declaration
};
program:: program (int x,int y)
//constructor defined
{
m=x;
n=y;
}
Void program::display()
//member function defined
{
Cout<<"m is"<<m;
Cout<<"n is"<<n;
}
Void main()
{
Clrscr();
```

```
program         p(10,20);
//by calling constructor implicitly
program         p1= program (20,30);
//by calling constructor explicitly
p.display();
p1.display();
Getch();
}
o/p:__m is 10

     n is 20

     m is 20

     n is 30.____________.
```

WAP to display 2 integers using parameterized constructor

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

```
class integer
{
int m,n;
  public:
    integer(int a,int b)
    {
    m=a;
    n=b;
    }
    void display()
    {
    cout<<"M="<<m<<endl;
    cout<<"N="<<n<<endl;
    }
};
  void main()
  {
  clrscr();
  integer A(100,200);
  integer B(25,50);
```

```
 A.display();
 B.display();
 getch();
 }
```

Copy constructor:

A copy constructor takes a reference to an object of the same class as itself as an argument.

- ✓ Copying values of one object into another object is known as copy constructor.
- ✓ It is also optional to specify the definition of copy constructor in a class.
- ✓ We can copy values of one object into another object in two ways

i) By passing overloaded assignment operator (=)

Ex:

ob2=ob1; where ob1 and ob2 are objects.

ii) By passing source object as parameter

ex:

ob2(ob1);

ob1 → source object

ob2 → destination or target object

//write a c++ to programme to display copy constructor

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

```
int x;
public:
sample()
{
x=0;
}
sample(int n)
{
x=n;
}
sample(sample &s)
{
x=s.x;
}
void show(void)
{
cout<<endl<<"\t x:"<<x;
}
};
void main()
```

```
{
sample a;
clrscr();
cout<<endl<<"a:";a.show();
sample b(10);
cout<<endl<<"b:";b.show();
sample c(b);   //copy constructor
cout<<endl<<"c:";c.show();
sample d=c;   //initialization
cout<<endl<<"d:";d.show();
sample e(d);
cout<<endl<<"e:";e.show();
sample f=d;  //assining  object
cout<<endl<<"f:";f.show();
getch();
}
```

Note:

Keyword “this” is a pointer which is used for pointing current object.

this EXAMPLE:

wap to perform addition of two numbers using this

```
class sample
{
 Int a,b;
Public:
  sample(int a,int b)
{
 This->a=a;
This->b=b;
}
Void display()
{
Cout<<a+b;
}
};
```

```
Void main()
{
Sample obj(10,20);
Obj.display();
}
```

DESTRUCTORS:-

A destructor as the name implies is used to destroy the objects that have been created by a constructor.

- ✓ It is also a special member function which is used to destroy the object.
- ✓ It will be called when object comes out of the scope.

RULES FOR DECLARING DESTRUCTOR:

- ✓ Destructor name should be same as class name.
- ✓ It should be preceded by (~) tilde symbol
- ✓ It should be declared under public section
- ✓ It doesn’t have any return type
- ✓ It can be a virtual.
- ✓ We cannot overload destructor i.e. one class can have only on destructor.
- ✓ Destructor cannot be parameterized

Note:

- ✓ When we create pointer to an object constructor and destructor will not be used.
- ✓ If we want to invoke constructor we have to use ‘new’ operator for memory allocating and for deallocating ‘delete operator’ explicitly.

//destructor Example1:

```
#include<iostream.h>
#include<conio.h>
class testing
{
public:
testing(void)
{
cout<<endl<<"OBJECT
CREATED";//default constructor
}
~ testing()
{
cout<<endl<<"OBJECT
DESTRUCTED:";//destructor
getch();
}
};
```

```
void main()
{
clrscr();
testing t1;
cout<<endl<<"THANK YOU:";
getch();
}
```

//Destructor Example 2

```
#include<iostream.h>
#include<conio.h>
int count=0;
class sample
{
public:
sample(void)
{
count++;
cout<<endl<<count<<" OBJECT
CREATED";//default constructor
}
~ sample()
{
cout<<endl<<count<<" OBJECT
DESTRUCTED:";//destructor
getch();
count--;
}
};
void main()
{
clrscr();
sample s1;
{
sample s2;
}
cout<<endl<<"THANK YOU";
getch();
```

}

Inheritance:

C++ strongly supports the concept of reusability. The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

The mechanism of deriving a new class from an old one is called inheritance.

Old class →base class (or) parent class.

New class →derived class (or) child class.

Defining Derived Classes:

A derived class can be defined by specifying its relationship with the parent class in addition its own details.

The general form of defining a derived class is:

Syntax:

```
Class derived-classname   :   visibility-mode
parent-classname
   {
            ……………………//
            ……………………//
           …………………… //          members
of derived class
    };
```

The visibility mode is optional and, if present may be either private or public.

- The default visibility mode is private.
- Visibility mode specifies whether the

features of the base class are privately derived or publicly derived.

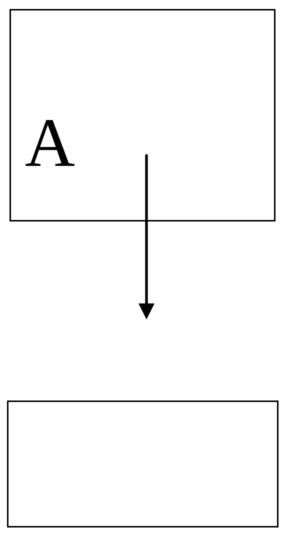
- When a base class is privately inherited by a derived class. Public members of the base class become private members of the derived class. Public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- When a base class is publicly inherited by a derived class. Public members of the base class become public members of the derived class. They are accessible to the objects of the derived class.
- In both cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

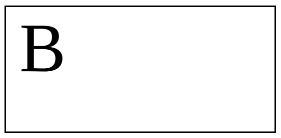
Types of inheritance:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

Single inheritance:-

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





->Write a c++ programme to display single inheritance

```
#include<iostream.h>
#include<conio.h>
class base
{
protected:
 int a,b,c;
 public:
 void getintegers(void)
 {
 cout<<endl<<"ENTER A B VALUES:";
 cin>>a>>b;
 }
 void showintegers(void)
 {
 cout<<endl<<"A :"<<a;
 cout<<endl<<"B: "<<b;
 }
 };
//class derive:protected base     //error
//class derive:private base       //error
class derive:public base
{
private:
float x,y;
public:
void getfloats(void)
{
```

```
cout<<endl<<"ENTER X,Y FLOAT
VALUES:";
cin>>x>>y;
}
void showfloats(void)
{
cout<<endl<<"X:"<<x<<"\t Y:"<<y;
}
};
void main()
{
base b1;
clrscr();
b1.getintegers();
b1.showintegers();
derive d1;
d1.getfloats();
```

```
d1.showfloats();
d1.getintegers();
d1.showintegers();
getch();
}
```

/*write a program to create a base class "person"having members personname, age,city,phonenumber.create a derived class physical_fit having menbers height and weight.accept two person details from input and display it. */

```
#include<iostream.h>
#include<conio.h>
class person
{
protected:
  char pname[20],city[10];
 int age;
 long int phno;
 public:
void getperson(void)
 {
 cout<<endl<<"ENTER PNAME,AGE:";
 cin>>pname>>age;
 cout<<endl<<"ENTER CITY AND
PHONE NO:";
 cin>>city>>phno;
 }
 void showperson(void)
 {
 cout<<endl<<"PNAME :"<<pname;
 cout<<endl<<"AGE: "<<age;
 cout<<endl<<"CITY:"<<city;
 cout<<endl<<"PHONE NO:"<<phno;
 }
 };
class physical_fit:public person
{
private:
float height,weight;
public:
void getdata(void)
{
cout<<endl<<"ENTER HEIGHT AND
WEIGHT:";
cin>>height>>weight;
}
void showdata(void)
{
cout<<endl<<"HEIGHT:"<<height;
```

```
cout<<endl<<"WEIGHT:"<<weight;
}
};
void main()
{
physical_fit p1,p2;
clrscr();
cout<<endl<<"ENTER FIRST PERSON
DETAILS----------------";
p1.getperson();
p1.getdata();
cout<<endl<<"ENTER SECOND PERSON
DETAILS---------------";
p2.getperson();
p2.getdata();
cout<<endl<<"PRESS ANY KEY TO
CONTINUE------------------";
```

```
getch();
clrscr();
cout<<endl<<"FIRST PERSON DETAILS--
--------------";
p1.showperson();
p1.showdata();
cout<<endl<<"SECOND PERSON
DETAILS---------------";
p2.showperson();
p2.showdata();
getch();
}
```

Multiple inheritance:-

A class can inherit the attributes of two or more classes is known as multiple inheritance.

Syntax of a derived class with multiple base classes is as follows.

```
Class derived : visibility  baseclass1 ,
visibility  baseclass2 , ………
{
……………;
……………;
};
```

//multiple inheritance Example.

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

```
{
protected:
    int rno,s1,s2,s3;
   char sname[20],city[10];
   public:
void getstudent(void)
{
cout<<endl<<"ENTER ROLLNO:";
cin>>rno;
cout<<endl<<"ENTER NAME AND
CITY:";
cin>>sname>>city;
cout<<endl<<"ENTER S1,S2 AND S3
MARKS:";
cin>>s1>>s2>>s3;
}
void showstudent(void)
```

```
{
cout<<endl<<"\n\t ROLLNO:" <<rno;
cout<<endl<<"\t SNAME:"  <<sname;
cout<<endl<<"\t CITY:"   <<city;
cout<<endl<<"\t S1:"<<s1<<"\t
S2:"<<s2<<"\tS3:"<<s3;
}
};
class sports
{
protected:
char gname[20];
int rank;
public:
void getgame(void)
{
cout<<endl<<"ENTER GAME NAME:";
cin>>gname;
cout<<endl<<"ENTER RANK:";
cin>>rank;
}
void showgame(void)
{
cout<<endl<<"\t GAME NAME:"<<gname;
cout<<endl<<"\t RANK:"<<rank;
}
};
//class result:protected student,protected
sports   //error
//class result:private student,private sports
//error
class result:public student,public sports
{
private:
```

```
int total;
float avg;
public:
void calc_total(void)
{
total=s1+s2+s3;
avg=(float)total/3;
}
void showall(void);
};
void result::showall(void)
{
showstudent();
cout<<endl<<"\t TOTAL:"<<total;
cout<<endl<<"\t AVERAGE:"<<avg;
showgame();
```

```
}
void main()
{
result r1;
clrscr();
r1.getstudent();
r1.getgame();
r1.calc_total();
clrscr();
r1.showall();
getch();
}
```

Ambiguity resolution in inheritance:-

When a function with the same name appears in more than one base class .

Ex:-

```
#include<iostream.h>
#include<conio.h>
class base1
{
 public:
  void display()
  {
   cout<<"\nbase1 display function";
  }
};
class base2
{
  public:
   void display()
   {
    cout<<"\nbase2 display function";
    }
};
class derived: public base1, public base2
{
 public:
  void display()
  {
   base1::display();
   base2::display();
   cout<<"\nderived display function";
   }
};
void main()
{
 clrscr();
 derived d;
 d.display();
 getch();
 }
```

Note:-

Ambiguity mat also arise in single inheritance applications.

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

```
class parent
{
 public:
   void show()
   {
    cout<<"\nbase class show method";
    }
};
class child: public parent
{
 public:
  void show()
  {
   cout<<"\nderived class show method";
   }
};
void main()
{
 child c;
 clrscr();
 c.show();
 c.parent::show();
 c.child::show();
 getch();
 }
```

Multilevel inheritance:-

The mechanism of deriving a class from another derived class is known as multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
    int rno;
   char sname[20],city[10];
   long int pincode,phno;
   public:
   void getstudent(void);
   void showstudent(void);
```

```
};
void student::getstudent(void)
{
cout<<endl<<"ENTER ROLLNO, 
SNAME:";
cin>>rno>>sname;
cout<<endl<<"ENTER CITY:";
cin>>city;
cout<<endl<<"ENTER PINCODE AND 
PHONE NUMBER:";
cin>>pincode>>phno;
}
void student::showstudent(void)
{
```

```
cout<<endl<<"\n\t ROLLNO:" <<rno;
cout<<endl<<"\n\t SNAME:"  <<sname;
cout<<endl<<"\n\t CITY:"   <<city;
cout<<endl<<"\n\t PINCODE:"<<pincode;
cout<<endl<<"\n\t PHONENO:"<<phno;
}
//class marks:protected student     //error
//class marks:private student      //error
class marks:public student
{
protected:
int s1,s2,s3;
public:
```

```
void getmarks(void)

{

cout<<endl<<"ENTER S1,S2,S3
MARKS:";

cin>>s1>>s2>>s3;

}

void showmarks(void)

{

cout<<endl<<"\t S1:"<<s1<<"\t
S2:"<<s2<<"\t S3:"<<s3;

}

};

//class result:protected marks  //error

//class result:private marks   //error
```

```
class result:public marks
{
private:
int total;
float avg;
public:
void calculate(void)
{
total=s1+s2+s3;
avg=(float)total/3;
}
void display(void);
};
```

```
void result::display(void)
{
showstudent();
showmarks();
cout<<endl<<"\t TOTAL:"<<total;
cout<<endl<<"\t AVERAGE:"<<avg;
}
void main()
{
result stu1,stu2;
clrscr();
cout<<endl<<"ENTER FIRST STUDENT
DETAILS----------------";
stu1.getstudent();
```

```
stu1.getmarks();
stu1.calculate();
cout<<endl<<"ENTER SECOND
STUDENT DETAILS----------------";
stu2.getstudent();
stu2.getmarks();
stu2.calculate();
cout<<endl<<"PRESS ANY KEY TO
CONTINUE:";
getch();
clrscr();
cout<<endl<<"FIRST STUDENT
DETAILS...........";
stu1.display();
```

```
cout<<endl<<"SECOND STUDENT
DETAILS...........";
stu2.display();
getch();
}
```

hierarchical inheritance:-

Derivation of several classes from one base class i.e. the features of one base class may be inherited by more than one class is called hierarchical inheritance.

```
//hierarchical inheritance
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person
```

```
{
protected:
char pname[20],city[10];
int age;
long int phno,pincode;
public:
void getperson(void)
{
cout<<endl<<"\t ENTER PNAME AND
AGE:";
cin>>pname>>age;
cout<<endl<<"\t ENTER CITY:";
cin>>city;
cout<<endl<<"\t ENTER PHNO AND
PINCODE:";
cin>>phno>>pincode;
}
```

```
void showperson(void)
{
cout<<endl<<"\t PNAME:"<<pname<<"\t
AGE:"<<age;
cout<<endl<<"\t CITY:"<<city;
cout<<endl<<"\t PHNO:"<<phno<<"\t
PINCODE:"<<pincode;
}
};
class student:public person
{
private:
int rno,total;
int marks[3];
float avg;
char grade[10];
public:
```

```
void getstudent(void);
void calc_grade(void);
void showstudent(void);
};
void student::getstudent(void)
{
cout<<endl<<"\t ENTER RNO:";
cin>>rno;
getperson();
cout<<endl<<"\t ENTER STUDENT
3SUBJECT  MARKS:";
for(int i=0;i<3;i++)
{
cin>>marks[i];
}
}
void student::calc_grade(void)
```

```
{
total=0;
for(int i=0;i<3;i++)
{
total+=marks[i];
}
avg=(float)total/3;
if(marks[0]<35||marks[1]<35||marks[2]<35)
strcpy(grade,"fail");
else
if(avg>=60)
strcpy(grade,"FIRST");
else
if(avg>=50)
strcpy(grade,"SECOND");
else
```

```
strcpy(grade,"THIRD");
}
void student::showstudent(void)
{
cout<<endl<<"\t ROLLNO:"<<rno;
showperson();
cout<<endl<<"SUBJECT MARKS:";
for(int i=0;i<3;i++)
{
cout<<"\t" <<marks[i];
}
cout<<endl<<"\t total:"<<total<<"\t
AVERAGE:"<<avg;
cout<<endl<<"\t GRADE:"<<grade;
}
class employee:public person
{
```

```
private:
int eno;
int basic;
float da,hra,tax,pf,grass,deduct,netsal;
public:
void getemp(void);
void calc_netsal(void);
void showemp(void);
};
void employee::getemp(void)
{
cout<<endl<<"\t ENTER EMPLOYEE
NUMBER:";
cin>>eno;
getperson();
cout<<endl<<"\t ENTER BASIC
SALARY:";
```

```
cin>>basic;
}
void employee::calc_netsal(void)
{
if(basic>=10000)
{
da=basic*0.30;
hra=basic*0.28;
pf=basic*0.17;
tax=basic*0.06;
}
else
if(basic>=5000)
{
da=basic*0.25;
hra=basic*0.18;
pf=basic*0.11;
tax=basic*0.03;
}
else
{
da=basic*0.10;
hra=basic*0.68;
pf=basic*0.05;
tax=basic*0.01;
}
grass=basic+da+hra;
deduct=pf+tax;
netsal=grass-deduct;
}
void employee::showemp(void)
{
```

```
cout<<endl<<"\t ENO:"<<eno;
showperson();
cout<<endl<<"\t BASIC:"<<basic;
cout<<endl<<"\t DA:"<<da;
cout<<endl<<"\t HRA:"<<hra;
cout<<endl<<"\t PF:"<<pf;
cout<<endl<<"\t TAX:"<<tax;
cout<<endl<<"\t GRASS:"<<grass;
cout<<endl<<"\t DEDUCT:"<<deduct;
cout<<endl<<"\t NETSALARY:"<<netsal;
}
void main()
{
clrscr();
student s1;
cout<<endl<<"ENTER STUDENT
DETAILS---------";
```

```
s1.getstudent();
s1.calc_grade();
s1.showstudent();
cout<<endl<<"PRESS ANY KEY TO
CONTINUE----------";
getch();
employee e1;
cout<<endl<<"ENTER EMPLOYEE
DETAILS-----------";
e1.getemp();
e1.calc_netsal();
e1.showemp();
getch();
}
```

hybrid inheritance:-

Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

Constructors with derived class Example:

```
#include<iostream.h>
#include<conio.h>
class alpha
{
int x;
      public:
       alpha(int i)
        {
        x=i;
        cout<<"alpha invoked\n"<<endl;
        }
        void show_a()
        {
        cout<<"x="<<x<<endl;
        }
        };

class beta
{
int y;
      public:
       beta(int j)
        {
        y=j;
        cout<<"beta invoked\n"<<endl;
        }
        void show_b()
        {
        cout<<"y="<<y<<endl;
        }
```

```
};

class set:public alpha,public beta
{
int m,n;
public:
set(int a,int b,int c,int
d):alpha(a),beta(b)
{
m=c;
n=d;
}
void display()
{
cout<<"m="<<m<<endl;
cout<<"n="<<n<<endl;
}
```

```
};
void main()
{
clrscr();
set s(10,20,30,40);
s.show_a();
s.show_b();
s.display();
getch();
}
```

POLYMORPHISM

Poly → many

Morphism → forms

Ability to take more than one form is called as polymorphism. There two type of polymorphism. They are

1) Static polymorphism
2) Dynamic polymorphism

1) Static polymorphism

Compiler time polymorphism can be implemented by the overloaded functions and operators. In compiler time polymorphism the compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding (or) static binding (or) static linking. Early binding means that an object is bound to its function call at compile time.

Let us consider a situation where the function name and prototype is same in both base and derived class.

- operator overloading
- Method overloading
- Method overriding

Dynamic Polymorphism:

Let us consider a situation where the function name and prototype is same in both base and derived class.

At this time is appropriate member function could be selected while the program is running this is known as run time polymorphism this can be implemented by using virtual function.

→ virtual function

Defining operator overloading:

Changing the functionality of an existing operator or adding functionality to an existing one is known as operator overloading.

Syntax:

```
    Returntype classname::operator op()
    {
     Function body                    //task
defined
```

```
      }
```

Note: op is the operator being overloaded. The op is preceded by the keyword **operator**. **Operator** op is the function name.

1) Unary operators

2) Binary operators

- ✓ The following operator cannot be overloaded.

  → sizeof

  → : :

  →: :* → member to pointer

- → ?:
- → .

SYNTAX FOR OPERATOR FUNCTION:

```
Returntype operator < operator symbol > (args)
 {
 --------------------------;
 --------------------------;
}
```

✓ Operator is a keyword and operator <operator symbol> is called operator function.

Ex:

```
    void display()                    void
operator +()
    {                                   {
      ____________________
______________________
      ____________________
______________________
     }                                   }
```

1. UNARY OPERATOR

i) if we are overloading unary operator by using member function then operator function should not contain any parameter.

ii) if we are overloading by using friend function, it should contain at least on object as a operator.

Ex:

```
#include<iostream.h>
#include<conio.h>
Class abc
{
  Int x;
  Int y;
//private data members decarations
  Int z;
  Public:
   Void getdata(int,int,int);
//member function declaration
   Void display();
   Void operator-();
//overloaded unary minus
};
//……….member functions
defined…………………………………
Void abc::getdata(int a,int b,int c)
{
X=a;
Y=b;
Z=c;
}
Void abc::display()
{
Cout<<”x is”<<x;
```

```
Cout<<"y is"<<y;
Cout<<"z is"<<z;
}
Void abc::operator-()
{
X=-x;
Y=-y;
Z=-z;
}
Void main()
{
 Clrscr();
Abc c;
c.getdata(10,-20,30);
c.display();
-c;                    //activates operator-()
function
```

```
c.display();
getch();
}
o/p: x is:10
         y is:-20
        z is:30
        x is:-10
       y is:20
      z is:-30.
```

2) BINARY OPERATOR:

i) if we are overloading binary operator using member function then operator function should contain at least one parameter.

ii) if we are overloading by using friend function then operator function should have at least two objects as a parameter.

//overloading binary operators:-
//overloading '+' operator:-

```
#include<iostream.h>
#include<conio.h>
class complex
{
private:
float a,b;
public:
complex()
{
a=0;   //default constructor
b=0;
}
complex(float real,float imag)
{
a=real;
b=imag;
}
friend complex operator+(complex
c1,complex c2);
friend void show(complex c);   //friend
function
};
complex operator+(complex c1,complex c2)
{
complex c3;
c3.a=c1.a+c2.a;
c3.b=c1.b+c2.b;
return(c3);
}
void show(complex c)
{
cout<<"\t"<<c.a<<"+i"<<c.b;
}
```

```
void main()
{
complex A(10.5,20.5);
complex B(30.5,40.5);
complex C;
clrscr();
C=A+B;    //calling operator+() function
cout<<endl<<"A:";show(A);
cout<<endl<<"B:";show(B);
cout<<endl<<".................";
cout<<endl<<"C:";show(C);
cout<<endl<<".................";
getch();
}
```

//overloading unary operators:-

//over loading unary minus(-)operator:-

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

```
{
private:
int x;
int y;
int z;
public:
negation(int x,int y,int z)
{
this->x=x;
this->y=y;
this->z=z;
}
void operator-(void);
void display(void);
};
void negation::operator-(void)
{
x=-x;
y=-y;
z=-z;
}
void negation::display(void)
{
cout<<endl<<"\t X:"<<x<<"\t Y:"<<y<<"\t
Z:"<<z;
}
void main()
{
negation n1(10,-20,-30);
clrscr();
cout<<endl<<"BEFORE OVER
LOADING:";
n1.display();
-n1;    //calling operator -() function
cout<<endl<<"AFTER OVER
LOADING:";
n1.display();
getch();
}
```

//overloading post&pre increment operators:-
//over loading post&pre increment (++op) operator:-

```
#include<iostream.h>
#include<conio.h>
class index
{
int value;
public:
  index()
  {
  value=0;
  }
  int getindex()
  {
  return value;
  }
  void operator ++()
  {
```

```
  value=value+1;
  }
};
  void main()
  {
  index id1,id2;
  clrscr();
  cout<<"\n OBJECT1="<<id1.getindex();
  cout<<"\n OBJECT2="<<id2.getindex();
   id1++;
   ++id1;
   id2++;
   ++id2;
  cout<<"\n OBJECT1="<<id1.getindex();
  cout<<"\n OBJECT2="<<id2.getindex();
  getch();
  }
```

METHOD OVERLOADING:

Same function name with different

argument list is called as method overloading.
Example:

```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
int m,n;
float p,q;
public:
void view()
{
cout<<endl<<"1ST METHOD"<<endl;
}
void view(int x,int y)
{
cout<<endl<<"2ND METHOD"<<endl;
m=x;
n=y;
cout<<endl<<m<<endl;
cout<<endl<<n<<endl;
}
void view(float x,float y)
{
cout<<endl<<"3ND METHOD"<<endl;
p=x;
q=y;
cout<<endl<<p<<endl;
cout<<endl<<q<<endl;
}
};
void main()
{
sample x;
clrscr();
x.view();
x.view(10,20);
x.view(10.5f,20.5f);
```

```
//x.sample::view();
//x.sample::view(30,40);
getch();
}
```

//METHOD OVERRIDING:

Same function name with same arguments list is called as method overriding.

Example:

```
#include<iostream.h>
#include<conio.h>
class a
{
public:
void view()
{
```

```
cout<<endl<<"A CLASS";
}
};

class b:public a
{
public:
void view()
{
cout<<endl<<"B CLASS";
}
};

class c:public b
{
public:
void view()
{
cout<<endl<<"C CLASS";
}
```

```
};

void main()
{
a x;
b y;
c z;
clrscr();
x.view();
y.view();
z.view();
//x.a::view();
//y.b::view();
//z.c::view();
getch();
}
```

Example2 :

Write a programme to display method overriding using pointer object.

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
void show()
{
cout<<"IN A"<<endl;
}
};
class B:public A
{
public:
void show()
{
cout<<"IN B"<<endl;
}
};
void main()
{
```

```
A a;
B b;
clrscr();
//a.show();//IN A
//b.show();//IN B
 A *ptr;  //pointer object always invokes the
base class method only.
 ptr=&b;
 ptr->show();
 getch();
}
```

Dynamic polymorphism:
Virtual function:

- ✓ When we create any virtual function in a base class that will be maintained in a table called virtual function table or virtual function.
- ✓ When we create any new virtual

function that function will be appended to that table

- ✓ In virtual functions highest priority is given to base class function.
- ✓ Polymorphism is possible only with pointers
- ✓ If we are creating pointer object of base class (class A) and we are storing the reference of derived class normal object (Class B) and we are not specifying virtual keyword for base class function. So here if we are trying to invoke the function, compiler invokes base class function by default.
- ✓ In order to invoke derived class function by using base class pointer object we need to make base class function as virtual.

RULES FOR VIRTUAL FUNCTION

- The virtual functions must be member of some class
- They cannot be static member
- They are accessed by using object pointer
- A virtual function can be friend of another class
- The prototype of the base class version of a virtual function and all the derived class version must be identical.
- we cannot have virtual constructor, but we can have virtual destructors if a virtual function is defined in the base class, it need not necessarily redefined in the derived class

DYNAMIC BINDING

Function resolved (recognized) at run time is called as dynamic binding.

Ex: virtual function, inheritance

```
//virtual base classes
#include<iostream.h>
#include<conio.h>
class a
{
protected:
int a;
public:
void showa(void)
{
a=10;
cout<<endl<<"\t A:"<<a;
}
};
class b:virtual public a
{
protected:
int b;
```

```
public:
void showb(void)
{
b=20;
cout<<endl<<"\t B:"<<b;
}
};
class c:public virtual a
{
protected:
int c;
public:
void showc(void)
{
c=30;
cout<<endl<<"\t C:"<<c;
}
};
class d:public b,public c
{
```

```
protected:
int d;
public:
void display()
{
d=40;
showa();
showb();
showc();
cout<<endl<<"\t D:"<<d;
}
};
void main()
{
d d1;
clrscr();
d1.display();
getch();
}
```

//virtual function Example1

//VIRTUAL FUNCTION Example2.

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
//void show()
virtual void show()
{
cout<<"IN A"<<endl;
}
};

class B:public A
{
public:
void show()
{
cout<<"IN B"<<endl;
}
};

class C:public A
{
public:
void show()
{
cout<<"IN C"<<endl;
}
};

class D:public A
{
public:
void show()
{
cout<<"IN D"<<endl;
}
};
```

```
void main()
{
 A a;
 B b;
 clrscr();
// a.show();//IN A
// b.show();//IN B
 A *ptr;
 ptr=&a;
 ptr->show();
 ptr=&b;
 ptr->show();
 C c;
 ptr=&c;
 ptr->show();
 D d;
 ptr=&d;
 ptr->show();
 getch();
```

}

pure virtual function

A pure virtual function is a function declared in a base class that has no definition relative to the base class.

Class containing pure virtual functions can not be used to declare any objects of its own. Such classes are called as abstract base class

Any virtual function which doesn't have any instructions or where body is equal to zero is called as pure virtual function.

Ex:

Class A

{

Public:

Virtual void display()   //virtual void display()=0;

{

}

};

ABSTRACT CLASS OR INCOMPLE CLASS:

- ✓ Any class which is having atleast one pure virtual function is said to be abstract class.
- ✓ We cannot create normal object for abstract class
- ✓ We can only create pointer object
- ✓ It can be used in only base class

- ✓ When we are using an abstract class as base class we have to provide definition for pure virtual function

<u>//dynamic polymorphism</u>
<u>//w.a.p to find area of rectangle and area of triangle using abstract class</u>

```
#include<iostream.h>
#include<conio.h>
class figure
{
public:
virtual void read(void)=0;
virtual void calc_area(void)=0;
virtual void display(void)=0;
};
class rectangle:public figure
```

```
{
private:
float length,breadth,area;
public:
void read(void)
{
cout<<endl<<"ENTER
LENGTH,BREADTH:";
cin>>length>>breadth;
}
void calc_area(void)
{
area=length*breadth;
}
void display(void)
{
cout<<endl<<"AREA OF
RECTANGLE:"<<area;
}
};
```

```
class triangle:public figure
{
private:
float base,height,area;
public:
void read(void)
{
cout<<endl<<"ENTER BASE,HEIGHT:";
cin>>base>>height;
}
void calc_area(void)
{
area=0.5*base*height;
}
void display(void)
{
cout<<endl<<"AREA OF
RECTANGLE:"<<area;
}
};
```

```
/*void main()
{
figure *f1;
clrscr();
f1=new rectangle;
cout<<endl<<"RECTANGLE:";
f1->read();
f1->calc_area();
f1->display();
cout<<endl<<endl<<"TRIANGLE:";
f1=new triangle;
f1->read();
f1->calc_area();
f1->display();
getch();
} */
void main()
{
clrscr();
rectangle r1;
r1.read();
r1.calc_area();
r1.display();
triangle t1;
t1.read();
t1.calc_area();
t1.display();
getch();
}
```

<u>**Pointers:-**</u>

Pointer is a derived data type that refers to another data variable by storing the variables memory address rather than data.

Declaring and initializing pointers:-

Data-type   *pointer-variable;

Ex:-

```
        int    *ptr , a ;        // declaration
         ptr = & a ;             //initialization
```

Program:-

```
 #include<iostream.h>
#include<conio.h>
Void main()
{
  int   a , *ptr1 , **ptr2 ;
  clrscr();
  ptr1=&a;
  ptr2=&ptr1;
  cout<<" the address of a : "<<ptr1<<"\n";
  cout<<"the address of ptr1 :
"<<ptr2<<"\n\n";
```

```
   cout<<”after incrementing the address
values:\n\n”;
   ptr1+=2;
   cout<<”the address of a : “<<ptr1<<”\n”;
   ptr2+=2;
   cout<<”the address of ptr1 :
“<<ptr2<<”\n”;
  getch();
}
```

<u>Output:-</u>

```
The address of a : 0*8fb6fff4
The address of ptr1 : 0*8fb6fff2
After incrementing the address values:
The address  of a :
The address of ptr1 :
```

<u>Manipulation of pointers:-</u>

```
#include<iostream.h>
#include<conio.h>
Void main()
```

```
{
int a=10,*ptr;
clrscr();
cout<<”the value of a is=”<<a<<”\n”;
*ptr=(*ptr)/2;
cout<<”the value of a is=”<<(*ptr);
getch();
}
```

Arithmetic operations on pointers:-

```
#include<iostream.h>
#include<conio.h>
void main()
{
   int   num[] ={56,75,22,18,90};
   int  *ptr , i ;
   clrscr();
   cout<<” the array values are:\n”;
   for(i=0;i<5;i++)
       cout<<num[i]<<”\n”;
/* initializing the base address of  num to ptr
*/
     ptr = num ;
     cout<<”\n value of ptr    :”<<*ptr<<”\n”;
     ptr++;
     cout<<”value of ptr++ is
:”<<*ptr<<”\n”;
     ptr--;
     cout<<”value of ptr -- is=”<<*ptr<<”\n”;
     ptr=ptr+2;
     cout<<”value of ptr+2 is=”<<*ptr<<”\n”;
     ptr=ptr-1;
     cout<<”value of ptr-1 is=”<<*ptr<<”\n”;
    ptr+=3;
    cout<<”value of ptr+=3
is=”<<*ptr<<”\n”;
    ptr-=2;
    cout<<”value of ptr-2 is = “<<*ptr<<”\n”;
    getch();
}
```

<u>Output:-</u>

```
 The array values are:
      56
      75
      22
      18
      90
      value of ptr        :     56
      value of ptr++   :     75
      value of ptr--   :       56
      value of ptr+2   :     22
      value of ptr-1   :     75
      value of ptr+=3   :     90
      value of ptr-=2   :     22
```

<u>Ex:-</u>Program to print sum of even numbers of a given array using pointers

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

```
void main()
{
  int   a[10],n,i,*ptr;
  clrscr();
   cout<<"enter  array range";
   cin>>n;
   cout<<"enter array values";
   for(i=0;i<n;i++)
   cin>>a[i];
   ptr=a;
   int sum=0;
   for(i=0;i<n;i++)
    {
            if ( *ptr % 2 ==0 )
                  sum= sum+ *ptr;
            ptr++;
   }
  cout<<"\n sum of even numbers "<<sum;
getch();
}
```

Array of pointers:-

Ex:-

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void  main()
{
      int   i=0;
      char   *ptr[10] ={  “red” , “ black”
,”white”,”green”};
      char   str[25];
      clrscr();
      cout<<”enter your favorite color”;
      cin>>str;
      for(i=0;i<4;i++)
       {
            if ( !  strcmp(str , ptr[i]))
     {
        cout<<"your favorite color is
available here ";
                break;
     }
      }

if(i==4)
  cout<<"your favorite color is not available
";
getch();
}
```

Pointers to objects:-

We can create the objects using pointers and new operator as follows.

Syntax:-

```
      classname   *ptr = new  classname;
```

Then ptr can be used to refer to the members as follows.

```
     ptr → member;
```

we can also create an array of objects using pointers.

Syantax:-

classname *ptr = new classname[size];

Ex:-

Item *ptr = new item[10]; //array of 10 objects

Ex:-

```
#include<iostream.h>
#include<conio.h>
class item
{
 int code;
 float price;
 public:
   void getdata(int a,float b)
   {
    code=a;
    price=b;
   }
   void show()
   {
    cout<<"code :"<<code<<"\n";
    cout<<"price :"<<price<<"\n";
   }
};
const int size=2;
void main()
{
 clrscr();
 item *p=new item[size];
 item *d=p;
 int x,i;
 float y;
 for(i=0;i<size;i++)
 {
    cout<<"input  code and price for item
="<<i+1;
    cin>>x>>y;
    p->getdata(x,y);
    p++;
 }
 for(i=0;i<size;i++)
 {
   cout<<"item ="<<i+1<<"\n";
   d->show();
   d++;
 }
getch();
}
```

Pointers to derived objects:-

```
#include<iostream.h>
#include<conio.h>
Class bc
{
     Public:
          Int b;
          Void show()
          {
          Cout<<"b="<<b<<"\n";
          }
};
Class dc:public bc
{
     Public:
          Int d;
          Void show()
          {
          Cout<<"b="<<b<<"\n";
          Cout<<"d="<<d<<"\n";
          }
};
Void main()
{
bc  *bptr;                    //base pointer
bc  base;
bptr=&base;                   //base address
```

```
bptr->b=100;              //access bc member
via  base pointer
cout<<”base pointer points to the base
object”<<”\n”;
bptr->show();
//derived class
dc   derived;
bptr=&derived;             // address of derived
object
bptr->b=200;               // access dc  via
base pointer
bptr->d=300;               // won’t work
cout<<”base pointer now points to derived
object\n”;
bptr->show();

dc   *dptr;                           //derived type
pointer
dptr=&derived;
dptr->d=300;
```

cout<<"dptr is derived type pointer\n";
dptr->show();

//use base pointer to access the derived members

Syntax:-

((derivedclassname *)baseclass pointer)->derivedclass member;

Ex:-

((dc *)bptr)->d=400;
((dc *)bptr)->show();

getch();
}

TEMPLATES

A significant benefit of object oriented programming is reusability of code which eliminates redundant coding. The concept of reusability is again implementing by using template concept.

A template in C++ allows the construction of a family of template functions and classes to perform the same operation on different data types.

The template declared for functions are called function templates and those declared for classed are called class templates.

- ✓ Templates are used for generic programming.

GENERIC PROGRAMMING:

- ✓ In computer science, generics is a technique that allows one value to take different data types. The programming style emphasizing use of this technique is called generic programming.
- ✓ Template is used to generate generic classes or generic function
- ✓ When we are performing common functionality on different data types we have to use templates.
- ✓ There are two types of templates

1) Class Template
2) function template

FUNCTION TEMPLATE

syntax:

```
template < class templatename>
templatename functionname(args)
 {
     ----------------------------
     ----------------------------
     ---------------------------
 }
```

//FUNCTION TEMPLATES

/*w.a.p to create a function template and find the sum of integer numbers&sum of floating-point numbers*/

```
#include<iostream.h>
#include<conio.h>
template<class t>
t sum(t p,t q)
{
return(p+q);    //generic function
}
void main()
{
int a,b;
clrscr();
cout<<endl<<"ENTER A B
 VALUES(integers):";
cin>>a>>b;
int c=sum(a,b);
cout<<endl<<"SUM OF INTEGER
 NO:"<<c;
float x,y;
cout<<endl<<"ENTER X Y
 VALUES(floating values):";
cin>>x>>y;
float z=sum(x,y);
cout<<endl<<"SUM OF FLOATING
 NO:"<<z;
getch();
}
```

/*w.a.p to exchange(swapping)two integer numbers and two floating point numbers and two character using function templates.*/

```
#include<iostream.h>
#include<conio.h>
template<class t>
void swap(t &p,t &q)
{
t temp;
temp=p;
p=q;
```

```
q=temp;
}
void main()
{
int a,b;
clrscr();
cout<<endl<<"ENTER A B
 VALUES(integers):";
cin>>a>>b;
cout<<endl<<"BEFORE SWAPPING
 A:"<<a<<"\t B:"<<b;
swap(a,b);
cout<<endl<<"AFTER SWAPPING
 A:"<<a<<"\t B:"<<b;
float x,y;
cout<<endl<<"ENTER X Y
 VALUES(floating points):";
cin>>x>>y;
cout<<endl<<"BEFORE SWAPPING
 X:"<<x<<"\t Y:"<<y;
```

```
swap(x,y);
cout<<endl<<"AFTER SWAPPING
 X:"<<x<<"\t Y:"<<y;
char ch1,ch2;
cout<<endl<<"ENTER CH1 CH2
 VALUES(characters):";
cin>>ch1>>ch2;
cout<<endl<<"BEFORE SWAPPING
 CH1:"<<ch1<<"\t CH2:"<<ch2;
swap(ch1,ch2);
cout<<endl<<"AFTER SWAPPING
 CH1:"<<ch1<<"\t CH2:"<<ch2;
getch();
}
```

CLASS TEMPLATE

```
Template < class T>
Class classname
{
```

```
-------------------
------------------
}

-
```

/*w.a.p to create a class template and find some of two integer and floating poniter members */

```
#include<iostream.h>
#include<conio.h>
template<class T>
class summing
{
private:
T a,b,c;   //generic class
public:
summing(T a,T b)
{
this->a=a;
this->b=b;
cout<<endl<<"\t A:"<<a<<"\t B:"<<b;
}
T sum(void)
{
c=a+b;
return(c);
}
};
void main()
{
clrscr();
cout<<endl<<"INTEGER VALUES:";
summing<int>s1(10,20);
int k1=s1.sum();
cout<<endl<<"INTEGER SUM:"<<k1;
cout<<endl<<"\t FLOATING POINT
VALUES:";
summing<float>s2(10.31,20.21);
float k2=s2.sum();
```

```
cout<<endl<<"FLOATING POINTS
SUM:"<<k2;
getch();
}
```

//w.a.p to create a class template to find area of rectangle.

```
#include<iostream.h>
#include<conio.h>
template<class T>
class rectangle
{
private:
T d1,d2,area;   //generic class
public:
void read(void)
{
cout<<endl<<"ENTER D1 D2 VALUES:";
cin>>d1>>d2;
}
void calc_area(void)
```

```
{
area=d1*d2;
}
void show(void)
{
cout<<endl<<"AREA OF THE
RECTANGLE:"<<area;
}
};
void main()
{
rectangle<int>r1;
clrscr();
cout<<endl<<"ENTER INTEGER
VALUES ONLY:";
r1.read();
r1.calc_area();
r1.show();
rectangle<float>r2;
cout<<endl<<"ENTER FLOATING
```

```
VALUES ONLY:";
r2.read();
r2.calc_area();
r2.show();
getch();
}
```

<u>EXCEPTION HANDLING:</u>

EXCEPTION

- ✓ It is a runtime error which stops normal execution of our program.
- ✓ In order to handle rum time errors we have different exception handling methods

For example:

1) try
2) catch
3) finally
4) throw

Try

in this block we will place all the instructions / statements which are going to raise run time error

Catch

in this block we will handle run time errors which are being raised by its corresponding try block, or it will be executed only when runtime error encountered

Finally

this block will be executed irrespective of exception is raised or not. This block is optional.

Throw

it is a statement which is used to throw the specified exception depends on our requirement i.e exception raised by user.

Note: single try block can have multiple

catch blocks but only one finally block.
Single program can have multiple try blocks.
Note: database closing statements will be placed in finally

```
        //EXCEPTIONAL HANDLING
/*w.a.p to devide two integer numbers if the divisbel value is zero the raise
an excetion.*/
#include<iostream.h>
#include<conio.h>
void main()
{
int a,b;
float c;
clrscr();
cout<<endl<<"ENTER A VALUE:";
cin>>a;
cout<<endl<<"ENTER B VALUE:";
cin>>b;
try
{
if(b==0)throw "connot devide with zero:";
c=(float)a/b;
cout<<endl<<"C:"<<c;
}
catch(char *s)
{
cout<<endl<<"ENTER RAISED";
cout<<endl<<s;
}
cout<<endl<<"THE END";
}
```

FILES

In order to store large volumes of data, we need to use some devices such as floppy disk or hard disk. The data is stored in these devices using the concept of files.

A file is a collection of related data stored in a particular area on the disk.

Program can be designed to perform the read and write operations on these files.

It involves 2 kinds of data communication.

- Data transfer between the console unit and the program.
- Data transfer between the program and a disk file.

The I/O system of c++ handles file operations which are very much similar to the console input and output operation.

It uses file streams as an interface between the program and the files.

The stream that supplies data to the program is known as input stream.(or) the input stream extracts data from the file.

The stream that receives data from the program is know an output stream (or) the output stream inserts (or writes) data to the file.

C++ supports file manipulation in the form of stream objects.

The stream objects cin and cout are used to deal with the standard input and output devices.

these objects are predefined in iostream.h header file.

There are three classes for file handling:

- ifstream - for handling input files
- ofstream - for handling output files

fstream - for handling files on which both input and output can be performed.

The classes ifstream,ofstream,and fstream are designed to manage the disk files and their declaration exists in the header file fstream.h

filebuf: The class filebuf sets the file buffer to read and write.

AVAILABLE FUNCTIONS FROM STREAM CLASSES:

Ofstream → open(),close(),put(),write(),seekp(), tellp()

Ifstream → open(),close(),get(),getlline(),read(),seekg(),t ellg

fstream →above all functions

eof → reached end of file or not

fail () → file existing or not

EOF → compare characters

open():

This function used to open a specified file in a specified mode.

SYNTAX:

ofstream object

ifstream object . Open (“filename”, “filemode”);

fstream object

(filemode Optional for ofstream,ifstream)

<u>File modes</u>

file mode specifies the purpose for which the file is opened.

The file mode parameter can take one(or more) of such constants defined in the class ios.

<u>parameter</u>	**<u>meaning</u>**
ios::app	Append to end-of-file
ios::ate	go to end- of -file on opening
ios::binary	binary file

ios::in open file for reading only

ios::out open file for writing only

ios::trunc delete contents of the file if it exits

close():

✓ This function used to close a file.

SYNTAX:

< object > . Close()

Note: object can be of ofstream class or ifstream class or fstream class.

Note: it is compulsory to close a file in C++ otherwise it doesn’t perform any operation on file

Ex1:

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

void main()
{
char name[20];
float cost;
clrscr();
ofstream outf("ITEM.txt");
cout<<"enter item name:";
cin>>name;

outf<<name<<"\n";

cout<<"enter item cost:";
cin>>cost;
outf<<cost<<"\n";

outf.close();


ifstream inf("ITEM.txt");
inf>>name;
inf>>cost;
cout<<"\n item name:"<<name<<endl;
cout<<"\n item cost:"<<cost<<endl;

inf.close();
getch();

}
```

Ex2:

```
#include<fstream.h>
```

```
#include<conio.h>
#include<iostream.h>
void main()
{
int num1=530;
float num2=1050.25;
clrscr();
ofstream outfile("number.bin",ios::binary);
outfile.write((char *) &num1,sizeof(num1));
outfile.write((char *) &num2,sizeof(float));
outfile.close();

ifstream infile("number.bin",ios::binary);

infile.read((char *) &num1,sizeof(int));
infile.read((char *) &num2,sizeof(num2));

cout<<"num1="<<num1<<"
"<<"num2="<<num2<<endl;
infile.close();
getch();
}
```

Ex3:

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<conio.h>
void main()
{
char ch;
  char filename[20];
  clrscr();
  cout<<"enter a filename"<<endl;
  cin>>filename;

  ifstream ifile(filename);
  if(!ifile)
  {
  cout<<"error opening"<<filename<<endl;
  exit(1);
  }
  else
  {
      while(ifile)
      {
      ifile>>ch;
      cout<<ch;
      }

    }
   getch();
   }
```

Ex4:

```
//w.a.p to print your program as output
#include<iostream.h>
#include<process.h>
#include<conio.h>
#include<fstream.h>
void main()
{
char ch,fname[12];
clrscr();
cout<<endl<<"ENTER FILE NAME:";
cin>>fname;
ifstream fin;
fin.open(fname);
if(fin.get(ch)==NULL)
{
cout<<endl<<"FILE DOES NOT EXIST";
getch();
exit(0);
}
while(!fin.eof())
{
fin.get(ch);
cout<<ch;
}
fin.close();
getch();
}
```

Ex5:

```
/*w.a.p accept data character by character
and store on a give file.again accept
the data from that file and display on a
screen*/
#include<iostream.h>
#include<stdio.h>
```

```
#include<conio.h>
#include<fstream.h>
void main()
{
char ch;
clrscr();
ofstream fout;
fout.open("test.txt");
cout<<endl<<"ENTER THE DATA UNTIL
PRESS F6 OR CTRL+Z:"<<endl;
while((ch=getch())!=EOF)
{
fout.put(ch);
}
cout<<endl<<"\n \t file copied";
getch();
cout<<endl<<"press any key to continue-----
-";
fout.close();
ifstream fin("test.txt");
cout<<endl<<"\n READ THE DATA
FROM FILE------\n \n";
while(!fin.eof())
{
fin.get(ch);
cout<<ch;
}
fin.close();
getch();
}
```

put() and get() functions:-

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

Syntax:-

get():-

```
    cin.get(variable);
```

(or)
variable=cin.get();

put():-

```
    cout.put(variable);
```

Ex:-

```
#include<iostream.h>
#include<conio.h>
void main()
{
int    count=0;
char c;
clrscr();
cout<<"enter input text";
cin.get(  c  );
while(c != '\n' )
{
     cout.put( c );
     count++;
     cin.get( c );
}
cout<<"\n number of characters are= "
<<count  <<"\n";
getch();
}
```

getline() and write() functions:-

getline():-

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

Syntax:-

```
      cin . getline ( line, size);
```

Ex:-

```
     char   name[20];
     cin.getline(name,20);
```

Ex:

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

void main()
      {
       clrscr();
       char s[80];
       cout<<"ENTER A STRING :";
```

```
        cin.getline(s,20);
        clrscr();
        cout<<"YOU HAVE ENTERED...";
        cout.write(s,10);
      }
```

write():-

This function displays an entire line on the output screen.

Syntax:-

```
  cout.write(line,size);
```

Ex:-

```
#include>iostream.h>
#include<conio.h>
void main()
{
 char  city[20];
clrscr();
cout<<"enter city name";
cin.getline(city,5);
cout<<city;
getch();
}
```

Note:-

In the above case, the input will be terminated after reading the following 4 characters.

Ex:-

```
#include<iostream.h>
#include<conio.h>
void main()
{
char  city[20];
clrscr();
cout<<”enter city name:\n”;
cin>>city;
cout<<” city name=”<<city<<”\n\n”;
cout<<”enter city name again: \n   ”;
cin.getline(city,20);
cout<<”city name now: “<<city<<”\n\n”;
cout<<”enter another city name: \n”;
cin.getline(city,20);
cout<<”new city name is=”<<city<<”\n”;
getch();
}
```

Ex:-

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

```
char  *s=”programming”;
clrscr();
int m=strlen(s1);
for(int i=1;i<=m;i++)
{
    cout.write(s,i);
    cout<<”\n”;
}
getch();
}
```

<u>Width():-</u>

We can use the width() function to define the width of a field necessary for the output of an item. Since it is a member function, we have to use an object to invoke it.

<u>Syntax:-</u>

cout.width(w);

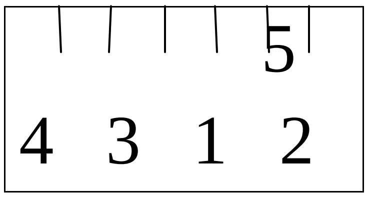
where w is the field width(number of columns).

Ex:-

cout.write(5);

cout<<543<<12<<"\n";

o/p:-



```
        5
4  3  1  2
```

Ex:-

cout.width(5);

cout<<543;

cout.wdth(5);

cout<<12;

program:-

```
#include<iostream.h>
#include<conio.h>
void main()
{
int    item[4]={10,8,12,15};
int    cost[4]={75,100,60,99};
cout.width(5);
cout<<   ” items”;
cout.width(8);
cout<<    ” cost”;
cout.width(15);
cout<<”total value”<<”\n”;
int sum=0;
for(int i=0;i<4;i++)
{
    cout.width(5);
    cout<<items[i];
    cout.width(8);
    cout<<cost[i];
    int value=items[i]*cost[i];
    cout.width(15);
    cout<<value<<"\n";
    sum=sum+value;
}
cout<<" \ngrand total is=";
cout.width(2);
cout<<sum<<"\n";
getch();
}
```

<u>Precision():-</u>

By default, the floating numbers are printed with six digits after the decimal point. However we can specify the number of digits to be displayed after the decimal point while printing the floating point numbers .

Syntax:-

cout.precision(d);

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

Ex:-

cout.precision(3);

cout<<3.141234<<"\n";

cout<<2.500654;

o/p:-

fill():-

this function is used to fill the unused positions by any desired character.

Syntax:-

cout.fill(ch);

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

Ex:-

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

o/p:-

```

5     2     5       0
```