C++

- Procedure oriented programming:

POP basically consist of writing a list of instructions for the computer to follow and organising these instruction in groups known as functions.

We normally use a flowchart to organise these functions and represent the flow of control from one action to another action. While we concentrate on developement of fun, a very little attention is given to the data that are being used by various fun. Each fun may have its own local data and also have access to global data. In a large program it is very difficult to identify what data is used by which fun. Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because fun are action oriented and do not really correspond to element of problem.

Characteristic of POP:

1) Emphasis is on doing things.
2) Large Program are divided into small program called as fun.
3) Most of fun share global data.
4) Data move openly around freely fun to fun.
5) fun transform data from one form to another.
6) Employs top-down approach in program design.

Object oriented Programming : (OOP)

OOP is an approach to program organisation and developement that attempt to eliminate some of the bit false of conventional programming methods by incorporating best of structure Programming with several powerful new concepts. it is new way of organising and developing programs and has nothing to do any particular language.

- In OOP treat data as critical element in program development and does not allow flow freely around the system.
- It types data more to fun^n that operate on it
- And protect its from accidential modification from outside fun^n
- OOP allows decomposition of problems in no. of entity called object & building data & fun^n. around these object.

Characteristics of OOP :

i) Emphasis is on data rather than procedure.
ii) programs are divided into what are known as objects.
iii) Data structures are designed such that they characterise the objects.
iv) fun^n that operate on the data of an object are type together in data structure.
v) Data is hidden and can't be accessed by external fun^n.
vi) objects may communicate with eachotter through fun^n.
(vii) New data and fun^n can be easily added whenever necessary.
(viii) Follows bottom of approach in program design

Features of OOP :

(i) Object : Objects are basic runtime entities in an object oriented system They may represent a person, a place, a bank account, a table of data or any item that program has to handle. They may also represent user defined data vectors, time list

Class and objects :

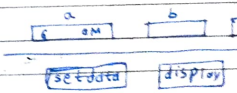
keyword — class name

```
class student
{
private:
    int rno;
    char nm[15];
public:
    void setdata()
    {
    cout<<"\n Roll No:";
    cin>>rno;
    cout<<"\n Name:";
    cin>>nm;
    }
```

member access specifier

member fun^n

student
private:
int rno;
char nm[15];
public:
void setdata()
void display()



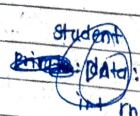
```
void display ()
{
cout << "\n Roll No: " << rno;
cout << "\n Name: " << nm;
}
};  // end of class student defination
void main ()
{
student a; // object creation of type student
a.setdata();
a.display();
}
```

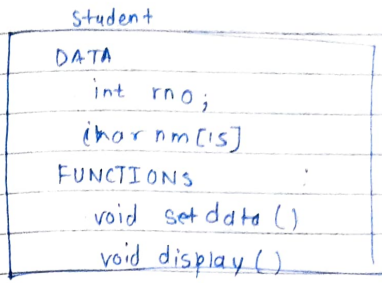
member function

member fun. calls

- feature of object oriented programming:

① Objects: objects are basic runtime entities in an object oriented system. They may represent a student, an account, a table of data or any item that program has to handle. Object take up and have associative address like structures in C. When a program is executed the object interacts by sending messages to one another. each objects contains data and code to manipulate data. objects can interact without having to know details of each others data or code.





② Class: The object contains data and code to manipulate data. the entire set of data and code of an object can be made user defined data type with the help of a class. objects are variables of the type class. once a class has been defined we can create any no. of objects belonging to that class. classes are user defined data types and behave like built in types of a programming language. i) The syntax used to create an object is no different ii) The syntax used to create an integer object in C. iii) Data abstraction and encapsulation.

③ Data Abstraction and encapsulation

The wrapping up of data and functions into a single unit is known as encapsulation. This is the most striking feature of a class. the data is not accessible to the outside world and only those functions which are wrapped in the class can access it these fun. provide an interface b/w objects data and the program. This insulation of the data from the direct access by the program is called data hiding or information hiding.

Abstraction refer to act of representing essential features without including background details or explanation. classes used concept of abstraction and defined as list of abstract attributes and functions to operate

on these attributes. They encapsulate all essential properties of the objects that are to be created. The attribute are sometimes called data member because they hold information. The functions that operate on these data are sometimes called member fun since, classes use concept of data abstraction, they are known as abstract data type.

④ Inheritance:

Inheritance is the process by which objects of one class acquire the property of another class the principal behind this sort of division is that each derive class shares common characteristics with class from which is derive.

The concept of inheritance provides the idea of reusability this means that we can add additional features to an existing class without modifying it. this is possible by deriving a new class from the existing one. The new class will have a combined features of both the classes. Note that each subclass defines only those features that are unique to it.

⑤ Polymorphism: This is another OOP concept. polymorphism means the ability to take more than one form. and operation may exibhit diff behaviours in diff. instances the behaviour depends upon the types of data used in the operation. Consider operation of addition for 2 no operation will generate sum if operands are string then operation would produce strings by concatanation. the process of making an operator to exibhit diff. behaviours in diff. instances is known as operator overloading.

Polymorphism plays an imp. role in allowing objects having diff. internal structures to share the same external interface.

⑥ Dynamic binding:

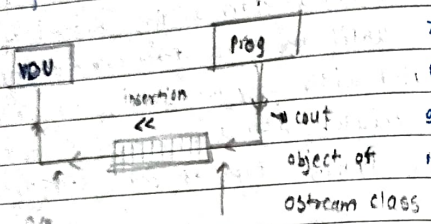
Binding refers to the linking of procedure call to the code to be executed in responce to the call. Dynamic binding is also known as late binding means code associated with a given procedure call is not known until the time of the call at runtime. it is associated with poly morphism and inheritance a fun call associated with a polymorphic reference depends on the dynamic type of that reference.

⑦ Message passing:

A message for an object is a request for a execution of a procedure therefore will invoke a fun in receiving object that generates the desired result. the message passing involves, the specifying name of object, name of fun (msg) and the information to be sent.

eg: Program to display "welcome" message.

```
#include <iostream.h>
void main()
{
 cout << "welcome";
 return;
}
```



cout <<
↑ insertion operator

eg:

```
void main()
{
 clrscr();
 cout << "\n Enter NO:";
 int x;
 cin >> x;
 cout << "\n NO:";
 cout << x;
 return;
}
```

list of files:

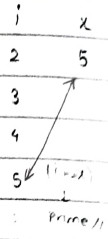
1) iostream.h
2) stdio.h
3) stdlib.h
4) string.h
5) iostream.h
6) iomanip.h
7) math.h
8) alloc.h
9) fstream.h
10) dos.h

eg: write a program to display table of inputed no.

```
void main()
{
 clrscr();
 cout << "\n Enter no:";
 int x;
 cin >> x;
 cout << "\n Table \n";
 for (int i=1; i<=10; i++)
 {
  cout << x*i;
  cout << " ";
 }
 return;
}
```

eg: write a program to check inputed no is prime no.

```
void main ().
{
 clrscr();
 cout <<"\n Enter NO:";
 int x;
 cin >>x;
 int i=2;
 while (i<x)
 {
  if (x%i==0)
    break;
  i++;
 }
  if (i==x)
    cout <<"\n prime";
  else
    cout <<"\n Not prime";
  return;
}
```



eg: i/p 10 no. and compute sum and average.

```
void main()
{
 clrscr();
 cout <<"Enter 10 no:";
 int x; int Sum=0;
 cin >>x;
 int i=1;
 for (i=0; i<10; i++)
 {
  do
  {
   cout <<" NO:";
   cin >>x;
   Sum += x;
  }
  while (++i<=10)
  cout <<"\n Total:";
  cout <<Sum;
  return;
 }
```

eg: Update and array of 10 integers and find out maximum value and minimum value.

```
void main
{
  clrscr();
  int x[10], i=0, max, min;
  while (i<10)
  {
    cout <<"\n NO: ";
    cin >> x[i];
    if (i==0)
      min = max = x[i];
    else
    {
      if (x[i]>max)
        max = x[i];
      else
      {
        if (x[i]<min)
          min = x[i];
      }
    }
    i++;
```

```
cout << "in Data \n" ;
for (i=0; i<10 ; i++)
{
  cout << x[i];
  cout << " " ;
}
Cout << "\n min : " << min
cout cin >> min ;
Cout "\n max : " << max ,
cin >> max ;
return ;
}
```

eg: write a program to implement sequential search for array of 10 integers.

```
void main ()
{
 clrscr ();
 int x[10], i=0 , SV ;
 while (i<10)
 {
  cout << "\n No:" ;
  cin >> x[i];
  i++;
 }
 cout << "\n Search Value : ";
 cin >> sv ;
 for ( i=0; i<10 ; i++ )
 {
```

SV

x	i	SV
2	0	
5		
3		
1		
4		

```
  if ( x[i] == SV )
    break ;
 }
 if ( i == 10)
   cout << "\n Not found ";
 else
   cout << "\n Found";
 return ;
}
```

- Cascading of insertion operator :

```
void main ()
{
 clrscr ();
 Cout << "Roll No:" << 1234 << "\n Name :" <<
      " Prasanna" << "\n Marks :" << 9.8
 return ;
}
```

O/P :

Roll No: 1234
Name : Prasanna
Marks : 9.8

In the above program we have used insertion operator repetatively. the statement first send the string "roll No:" to cout and then sends value of roll no. and then sends string "Name" and then string "Prasanna" and then sends string "marks" and then send data double "9.8".
The multiple use of insertion operator in one

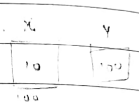
statement is called cascading.
when cascading and o/p operator we should ensure necessary blank spaces b/w blank items. we can also cascade an input operator extraction.

- Reference variables:

Syntax:

```
<data-type> & <ref> = <variable> ;
```

```
void main()
{
 int x=10;
 int &y = x;
 cout << "\n x : " << x ; // 10
 cout << "\n x : " << y ; // 10
 x+=5;
 cout<<"\n x :" << y ; // 15
 y+=2;
 cout << "\n x :" <<x ; // 30
}
```



C++ introduces a new kind of variable known as the Reference variable. a ref variable provides an alias (alternative name) for a previously defined variable.
A ref. var. must be initialized at the time of declaration this establishes the correspondence b/w reference and the data objects which it names.
it is imp. to note that the initialization of ref. variable is completely diff. from assignment to it.
a measure application of reference variable is in passing arguments to fun. such fun. calls are known as call by reference. The call by ref. mechanism is useful in object oriented programming. because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth.
it is also imp. to note that references can be created not only for built in types but also for user defined data types.

- Scope resolution operator: (::)

Syntax:

```
:: <variable.name>
```

```
int a = 1000 ;
void main ()
{
  int a = 1 ;
  {
    int a = 10 ;
    {
      int a = 100 ;
      cout << "\n A: " << a << "\t" << :: a ;
    }
    cout << "\n A: " << a << "\t" << :: a ;
  }
  cout << "\n A: " << a << "\t" << :: a ;
}
```

C++ is a block structured language. blocks and scopes can be used in constructing programs. we know that same variable name can be used to have diff. meaning in different blocks. the scope of the variable extends from the point of its declaration till the end of block containing the declaration the variable declared inside block is said to be local to the block. In C the global version of variable cannot be accessed within inner block. C++ resolves this problem by introducing a new operator (::) called the scope resolution operator. This can be use to uncover hidden variable. the Scope res. operator allows access to the global version of variable. a major application of scope res. operator is in the classes to identify the :: class to which a member fun belongs.

- Memory management operators:

Use of new and delete operators:

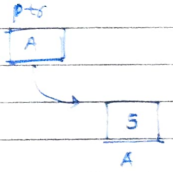
C uses malloc() and calloc() fun to allocate memory dynamically at runtime similarly it uses fun free to free dynamically allocated memory. we use dynamic allocation technique when it is not known in advance how much of memory space is needed although C++ supports this fun, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in the better and easier way. since these operators manipulate memory on the free store they are also known as free store operators. an object can be created using new and destroyed by using delete as and when required. a data object created inside block with new will remain in existence until it is explicitly destroyed by using delete. thus the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

(A) Use of new operator:

Syntax:

```
<pointer_variable> = new <data_type>;
```

```
void main()
{
  int *ptr;
  ptr = new int;
  cout << "\n Enter NO: ";
  cin >> *ptr;
  cout << "\n NO:" << *ptr;
  delete ptr;
  return;
}
```

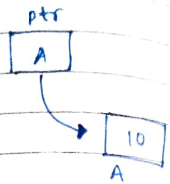


(B) Use of new operator:

Syntax:

```
<Pointer-variable> = new <data-type> (<initial value>);
```

```
void main()
{
 int *ptr;
 ptr = new int (10);
 cout<<"\n No: "<<*ptr;
 cout<<"\n Enter No: ";
 cin >> *ptr;
 cout<<"\n No: "<<*ptr;
 delete ptr;
 return;
}
```

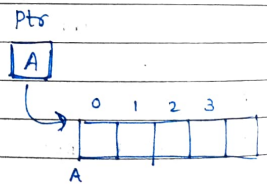


③ Use of new operator to allocate continuous memory.

Syntax:

```
< pointer_variable >.= new < data type > [ < no-of elements > ];
```

```
void main ()
{
 int *ptr;
 ptr = new int [5];
 int i=0;
 while (i<5)
 {
  cout<<"\n Enter No:";
  cin >> ptr [i];
  i++;
 }
 cout <<"\n Data \n";
 for( i=0; i<5 ; i++)
   cout << ptr [i] <<" ";
 delete [] ptr;
 return;
}
```



The new operator offers following advantages over the function malloc:

i) It automatically computes the size of the data objects. we need not use the operator sizeof().

ii) It automatically returns the correct pointer type so that there is no need to use the typecast.

iii) It is possible to initialize the object while creating the memory space.

iv) Like any other operator the new and delete can be overloaded.

- **Manipulators:**

The file to be included is iomanip.h

The manipulators are operators that are used to format the data object.

following are most used manipulators:

① **setw (n)**: sets column width for insertion.

② **setprecision (n)**: sets decimal places from fixed width.

③ **setfill (ch)**: sets filling character to fill blanks from the fixed width.

④ **endl**: inserts newline character.

```
void main()
{
clrscr();
cout<<12<<" "<<"sandeep"<<" "<<83.92<<"\n";
cout<<12345<<" "<<"Amita"<<" "<<63.97<<"\n";
return;
}
```

o/p:

1	2		s	a	n	d	e	e	p		8	3	.	9	2	0	0	0
1	2	3	4	5		A	m	i	t	a		6	3	.	9	7	0	0	0

Now

```
void main()
{
 clrscr();
cout << setw(5) << 12 << " " << setw(15)
     << setiosflags(ios::left) << "Sandeep"
     << resetiosflags(ios::left) << " " << setw(6)
     << setprecision(2) << setfill('*') << 83.6g <<
      setfill(' ') << endl;
cout << setw(5) << 12345 << " " << setw(15)
     << setiosflags(ios::left) << "Amita" << reset-
      iosflags(ios::left) << " " << setw(6) <<
     setprecision(2) << 63.97 << endl;
return;
}
```

		1	2		s	a	n	d	e	e	p							*	8	3	.	6	9
1	2	3	4	5		A	m	i	t	a									6	3	.	9	7

- **Functions in C++**

Functions play an important role in C program development. deviding a program in to fun^n is one of the major principles of top down structured programming the advantage of using function is that it is possible to reduce size of a program by calling and using them at different places in the program.

When the fun^n is called, control is transferred to the first statement in the fun^n body. the other statements in the fun^n body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. infact C++ has added many new features to fun^ctions to make them more reliable and flexible. like C++ operators a C++ fun^n can be overloaded to make it perform different tasks. depending on arguments passed on it.

In C++ the main fun^n returns the value of type int to the operating system. therefore explicitly defined new fun^n as int main. The fun^n that have return value should use the return statement for termination. mini operating system test the return value to determine if there is any problem. the normal convention is that an exit value of zero means the program ran successfully, while non-zero value means

there was a problem. the explicit use of return statement indicate that the program was successfully executed.

Function Prototyping :

This is one of the major improvements added to C++ functions. the prototype describes the fun^n interface to the compiler by giving details such as the no. and type of arguments and the type of return values. with fun^n prototyping a template is always used when declaring and defining a fun^n. when a fun^n is called the compiler uses the template to ensure that proper arguments are passed and the return value is treated correctly. any violation in matching the arguments or return type will be the caught by the compiler at the time of compilation itself. In a fun^n declaration the name of the arguments are dummy and they are optional. Ingeneral we can either include or exclude the variable names in the argument list of prototype. Variable names in the prototype just act as a placeholder and therefore if names are used they don't have to match the names used in the fun^n call in the fun^n definition.

Fun^n call by reference :

In traditional C a fun^n call passes arguments by value the called fun^n creates a new set of variables and copies the values of arguments into them. the fun^n does not have access to the actual variables in the calling program and can only work on the copies of values this mechanism is fine if the fun^n does not need to alter the values of original variables in the calling program.

But there may arise situations where we would like to change the values of variables in the calling Program. if the fun^n call by passing value mechanism is used then the changes would be local. Provision of the reference variable in the C++ permits us to pass parameters to the fun^n by reference. when we pass arguments by reference the formal arguments in the called function becomes aliases to the actual arguments in the calling fun^n. This means that if the fun^n is working with whole arguments it is actually working on the original data.

Function

eg: write fun^n to swap 2 values.

① Pass by value : (temperory change)

```
void main()
{
  int a, b;
  void swap (int, int); // by value
  cout <<"\n Enter 2 Nos:";
  cin >> a >> b;
  cout << "\n A:" << a << "\n B:" << b;
  swap (a, b);
  cout <<" A:"<< a << "\n B:" << b;
  return;
}
```

```
void swap (int p, int q)
{
int r = p;
p = q;
q = r;
}
```

② By reference: (Permanent change)

```
void main()
{
int a, b;
void swap (int &, int &), // by reference
cout << "Enter 2 no:";
cin>> a >> b;
cout << "\n A :" << a << "\n B :" << b,
swap (a, b);
cout << "\n A :" << a << "\n B :" << b,
return;
}
void swap (int &P, int &q)
{
int r = P;
P = q;
q = r;
}
```

- Inline fu^n:

One of the objective of using fu^n the program is to save some memory space which becomes appreciabl when fu^n became likely to be called many times. whenever everytime fu^n is called it takes lot of extra time in executing in series of instruction for task such as jumping to the fu^n, saving registers, pushing argumen -s into the stack and returning to the calling fu^n when a fu^n is small a substential percentage of execution time maybe spend in such over heads.

One solution to this problem to use macro definations Popularly known as macros. Preprocessor macros popular in C the major drawback with macros is that they are not really functions and therefore usual error message does not occur during compilation. C++ has diff sol^n to this problem. to eliminate cost of cause to small function C++ proposes a new feature called inline fu^n & inline fu^n is a fu^n that is expanded inline when it is invoked. i.e compiler replaces the fu^n call with corresponding fu^n defination. The inline fu^n is defined as follows

```
inline < fu^n header>
{
_____
_____
_____
}
```

Note: The inline keyword nearly sends a request, not a command to the compiler. The compiler may ignore this request if the fu^n def^n is too long or too complicated and compile the fu^n as a normal fu^n

Following are some of the situations where inline expansion may not work these are :

(1) For fun returning values if a loop, a switch or a goto exist.

(2) For fun not returning values if a return statement exist.

(3) If fun contains static variables.

(4) If inline fun are recursive.

eg:

```
inline int Sqr(int n)
{
 return n*n;
}

inline int cube(int n)
{
 return n*n*n;
}

inline void display()
{
 cout << "Welcome";
}

void main()
{
 display();
 cout << sqr(7) << endl;
 cout << cube(7) << endl;
 return;
}
```

o/p

Welcome
49
343

- **Default Arguments:**

C++ allows us to call a fun without specifying all its arguments. In such cases the fun assigns a default value to the parameter which does not have matching argument in the fun call. default values are specified when the fun is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. The default value is specified in a manner syntatically similar to a variable initialization. A default argument is checked for type at time of declaration and evaluated at the time of call. One importat pt. is to note that only the trailing arguments can have default values and therefore we must add defaults from right to left. We cannot provide default value to particular argument in the middle of an argument list.

Default arguments are useful in situations where some arguments always have the same value.

Advantages:

1) We can use default arguments to add new parameters to the existing fun.
2) Default arguments can be used to combine similar fun into one.

eg:

```
void repli (char ch, int n = 5)
{
 cout << "\n";
 for (int i=1; i<=n; i++)
  cout << ch;
}
void main()
{
 repli ('A');   // A A A A A
 repli ('#',3); // # # #
 return;
}
```

- C-style fu^ with default argument.

```
void main()
{
 void repli (char ch, int n=5);
 repli ('A');
 repli ('#',3);
 return;
}
void repli (char ch, int n=5);
{
 printf ("\n");
 for (int i=1; i<=n; i++)
  printf (ch);
}
```

- Function Polymorphism and Overloading:

Overloading refers to the use of the same thing for different purpose. C++ also permits overloading of a functions this means that we can use the same function name to create functions that perform variety of different task this is known as fu^ polymorphism in OOP.

Using the concept of function overloading we can design a family of functions with one fu^ name but with diff. argument list. The fu^ would perform different operations depending on the argument list in the fu^ call. The correct fu^ to be invoked is determined by checking the number and type of arguments but not on the fu^ type.

A fu^ call first matches the prototype having the same number and type of arguments and then calls the appropriate fu^ for execution a best match must be unique. The fu^ selection involves the following steps:

① The compiler first tries to find and exact match in which the types of actual arguments are the same. and use that fu^

② If an exact match is not found, the compiler uses integral promotions to an actual arguments. such as:

ⓐ char to int.

ⓑ float to double

to find a match.

ⓒ

③ when either of them fails the compiler tries to use the built in conversions to the actual arguments and then uses the fu: whose match is unique If the conversion is possible to have multiple matches then the compiler will generate and error message suppose we use following two functions.

```
void display (long);
void display (double);
```

and a function call.

```
display (10);
```

will cause an error message because int argument can be converted to long or double, thereby creating an ambiguous situation as which version of display should be use.

④ If all of the steps failed then the compiler will try the user defined conversion in combination with integral promotions and built in conversions to find a unique match. User defined conversions are often used with handling last objects.

```
void display ()
{
  cout << "\n Welcome";
}
void display (int a)
{
  cout << "\n Data :" << a;
}
void display (double d)
{
  cout << "\n Data: " << d;
}
```

```
main()
void display()
{
  display ();   // welcome
  display ('A');   // 65
  display (9.2f);
}
```

- Class and Objects:

C structure	C++ struct

① Defination
struct <tag_name>
{
——;
——;
——;
};
② variable creation
struct <tag-name> <variable>;
③ operator . and →
④ only placeholders are allowed, fu": are not allowed.
⑤ No security to data element
⑥ P=q+5; where P & q are variables of struct gives error.
	① struct < tag_name>
	{
	data elements
	fu": defination;
	};
	② <tag_Name> <variable>;
	③ operators .& →
	④ both data elements & fu": defination are allowed
	⑤ By default all elements are public & can be declared private.
	⑥ operator behaviour can be defined.
	⑦ size of(tag_Name)
	⑧ allows to apply all OOp features.

The unique feature of 'C' is structures provides a method for packing together data of different ~~datatu~~ types. structure is a convinient tool for handling a group of logically related data items it is a user defined data types. once a structure type has been defined we can create variable of that type using declarations that are similar to built in types.

Limitation of C structure:

① The C does not allow struct datatype to be treated like built in types. we can not operate arithmetic or relational operators over they do not permit data hiding. ie. all members of structure are public.

Extension to structures:

C++ supports all the features of structures as defined in C. C++ has expanded its capabilities further to suit its OOp Phylosopy. It attempts to bring user defined types as close as possible to the built in types and also provides the facility to hide the data which is one of the principles of OOP. inheritance is also supported by C++.

In c++, a structure can have both variables and function as members. it can also declares sum of its members as private so that they can not be accessed directly by the external function. In c++ the structure names are stand alone and can be used like any other type names In other words struct keyword can be omitted in the declaration of structure variables

C++ incorporates all these extensions in user defined type known as class. there is very little Syntatical difference b/w structures and classes in C++ and therefore they can be used interchangably with minor modifications. Since, Class is a specially introduced data type in C++. most of the C++ programmers tend to use the structures for holding only data and classes to hold both data functions. Therefore, we will not discuss structures further anymore.

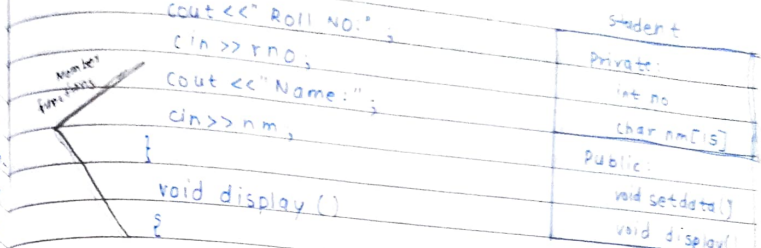
Class:

Class is a way to bind data and associated functions together. It allows the data and functions to be hidden, if necessary, from external use. When defining a class we are creating a new abstract data type that can be treated like any other built in data type. A class specification has two parts:

1) Class declaration.
2) Class function definations.

Eg: Program to implement Class Student.

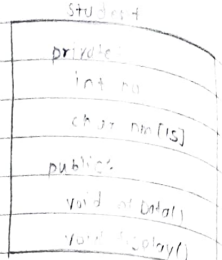
```
Class student
{
  private:
    int rno;
    char nm[15];
  Public:
    void setData()
    {
      cout << "Roll NO.";
      cin >> rno;
      cout << "Name:";
      cin >> nm;
    }
    void display()
    {
      cout << "\n Roll NO:" << rno;
      cout << "\n Name:" << nm;
    }
};
Void main()
{
  student a;
  a.setData();
  a.display();
  return;
}
```



- Member fun: definations outside the class:

```
Class student
{
  private:
    int rno;
    int nm;
  public:
    void setData();
    void display();
};
```

```
void student :: setData()
{
cout << "\n Roll No:";
cin >> rno;
cout << "\n Name:";
cin >> nm;
}
void student :: display()
{
cout << "\n Roll No:" << rno;
cout << "\n Name:" << nm;
}

void main()
{
student a;
a.setData();
a.display();
return;
}
```



eg: Create a header file with name "student.h" and define class.

```
class student
{
int rno;
char nm[15];
double mrk;
Public:
void setData()
{
cout << "\n Roll no:";
cin >> rno;
cout << "\n Name:";
cin >> nm;
cout << "\n Marks:";
cin >> mrk;
}
void setData(int no)
{
rno = no
cout << "Name:";
cin >> nm;
cout << "Marks:";
cin >> mrk;
}
void setData(int no, char *m)
{
rno = no;
strcpy(nm, m);
cout << "Marks:";
cin >> mrk;
}
void display()
{
cout << "\n Data: \n Roll No: " << rno << "\n Name: " << nm
<< "\n Marks:" << mrk;
}
```

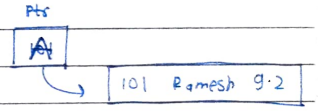
```
int getNO () { return rno ; }
double getmarks () { return mrk ; }
char * getName () { return nm ; }
void setMarks (double d)
{
  mrk = d ;
}
};
```

* store this with name "Student.h"

1)
```
# include "Student.h"
void main ()
{
  Student a ;
  a.setData ();
  a.Display ();
  Student b ;
  b.setData (120) ;
  b.display ();
  Student c ;
  c.setData (101, "Ganesh");
  cout << "\n Data : \n Roll No : " << c.getNO ()
       << "Name : " << ~~c.getMarks~~ c.getName()
       << "\n Marks : " << e.getMarks ();
}
```

2)
```
// Use of class Pointer to invoke member fu?
# include "student.h"
void main ()
{
  Student a ;
  Student * ptr ;
  ptr = &a ;
  ptr → setData (101) ;
  ptr → display ();
  return ;
}
```

3)
```
// Use of New operator to create instance

# include "student.h"
void main ()
{
  Student * ptr ;
  clrscr ();
  ptr → setData (101);
  ptr → display ();
  delete ptr ;
  return ;
}
```



Q)

i) Use of array as member of class :
ii) Member fn definations outside the class
iii) Nesting of functions

A) Create a header file with "array.h" and implement class:

```
#define M 5
class Array
{
  int x[M];
public:
  void init();
  void setdata();
  void setdata(int *);
  void setdata(Array &);
  void display();
  int calctot();
  int calcavg()
  {
    return Calctot()/M;
  }
  int findMax();
  int findMin();
  int search(int);
  int isOrdered();
  void findRepl(int, int);
};
```

```
void Array::init()
{
  int i=0;
  while(i<M)
  {
    x[i] = 0;
    i++;
  }
}
void Array::setdata()
{
  cout<<"Data:\n";
  for(i=0; i<M; i++)
    cin>>x[i];
}
void Array::Setdata(int *a)
{
  for(int i=0; i<M; i++)
    x[i] = a[i];
}
void Array::setdata(Array &t)
{
  for(int i=0; i<M; i++)
    x[i] = t.x[i];
}
int Array::Calctot()
{
  int tot = x[0], i=1;
  while(i<M)
    tot += x[i++];
  return tot;
}
```

```
void Array :: display ()
{
  cout << "\n Data: \n";
  for (int i=0; i<M; i++)
    cout << setw (5) << x[i];
}
int Array :: findMax ()
{
  int max = x[0], i=1;
  while (i<M)
  {
    if (x[i] > max)
      max = x[i];
    i++;
  }
  return max;
}
int Array :: findMin()
{
  int min = x[0]; i=1;
  while (i<M)
  {
    if (x[i] < min)
      min = x[i];
    i++;
  }
  return min;
}
```

x	i	max
1	0	1
2	1	2 ..
4	2	4
5	3	5 ..
~~3~~	~~4~~	
5	5	

x	i	min
2	0	2
5	1	1
3	2	
1	3	
8	4	

```
int Array :: search (int sv)
{
  int i=0;
  while (i<M)
  {
    if (x[i] == sv)
      break;
    i++;
  }
  return (i != M);
}
void Array :: findrepl (int sv, int rv)
{
  int i=0;
  while (i<M)
  {
    if (x[i] == sv)
      x[i] = rv;
    i++;
  }
}
int Array :: isOrdered ()
{
  int i=1;
  while (i<M)
  {
    if (x[i] < x[i-1])
      break;
    i++;
  }
  return (i == M);
}
```

x	i
2	0
1	1
3	2
4	3
5	4

* save and close the file "Array.h"

①
```
# include "Array.h"
void main()
{
 Array P;
 p.setdata();
 p.display();
 Array q;
 q.setdata(p);
 q.display();
 cout << "\n Total : " << q.calctot();
 cout << "\n Avg : " << q.calcavg();
}
```

- Nesting of Objects :

```
# include "Array.h"
Class Result
{
 int rno;
 Array mrk;
Public:
 Void setResult()
 {
  cout << "\n Roll No : ";
  cin >> rno;
  cout << "\n Marks : ";
  mrk.setdata();
}
void setResult(int a)
{
 rno = a;
 cout << "\n Marks : "
 mrk.setdata();
}
int getNo()
{
 return rno;
}
int getTotal()
{
 return mrk.calctot();
}
int getAvg()
{
 return mrk.calcavg();
},
void display()
{
 cout << "\n Roll No : " << rno;
 cout << "\n Marks : ";
 mrk.display();
 cout << "\n Total : " << getTotal();
 cout << "\n Avg : " << getAvg();
}
};
```

```
void main()
{
 Result a;
a.setResult();
a.display();
Result b;
b.setResult(121);
b.display();
return;
}
```

- **Friend Functions:**

We have been emphasizing that the private members cannot be accessed from outside the class i.e. a non-member function cannot have an access to the private data of a class. however there could be a situation where we would like 2 classes to share a particular function in such situations c++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function friendly to a class, we have to simply declare this fun. as a friend of the class as shown below:

```
class ABC
{
    ____;
    ____;
 Public:
    ____;
    ____;
    friend void display (ABC &);
};
```

The function declaration should be preceded by the keyword friend. The fun. is defined elsewhere in the program like a normal c++ function. The function def. does not use either the keyword friend or scope resolution operator. The functions that are declared with keyword friend are known as friend functions. a function can be declared as a friend in any no. of classes. A friend function, although not a member function, has full access rights to the private members of the class.

characteristics:

1) It is not in the scope of the class to which it has been declared as friend.
2) since it is not in the scope of the class it cannot be called using objects of that class.
3) It can be invoked like a normal function without the help of any object.
4) unlike member functions it can not access member names directly and has to use an object name and

dot membership operator (.) with each member name.
5) It can be declared either in the public or private class without affecting its meaning.
6) Usually it has the objects as arguments.

eg:
```
Class Number
{
  int no;
public:
  void setdata()
  {
  cout <<"\n Enter NO:";
  cin>> no;
  }
  void display()
  {
   cout <<"\n NO:"<<no;
  }
  friend void update (Number &, int n);
};

void update (Number &K, int n)
{
  K.no = K.no + n;
}
void main()
{
  Number t;
  t.setdata();
  t.display();
  update (t, 5);
  t.display();
}
```

o/p
```
Enter NO: 10
NO : 10
```

eg: A regular fun friend for both classes:
```
Class B; // forward declaration
Class A
{
  int a;
public:
  void set()
  {
   cout <<"\n A a:";
   cin >> a;
  }
  void display()
  {
   cout <<"\n A a:"<<a;
  }
  friend void swap (A&, B&);
};

Class B
{
  int b;
public:
  void set()
  {
   cout<<"\n B b:";
   cin >> b;
  }
  void display()
  {
   cout<<"\n B b:"<< b;
  }
  friend void swap (A&, B&);
};
```

dot membership operator(.) with each member name.
5) It can be declared either in the public or private class without affecting its meaning.
6) Usually it has the objects as arguments.

eg:
```
class Number
{
  int no;
public:
  void setdata ()
  {
   cout <<"\n Enter NO:";
   cin >> no;
  }
  void display ()
  {
   cout << "\n No:" << no;
  }
  friend void update (Number &, int n);
};

void update (Number &K, int n)
{
  K.no = K.no + n;
}
void main ()
{
  Number t;
  t.setdata ();
  t.display ();
  update (t, 5);
  t.display ();
}
```

o/p
```
Enter NO: 10
No : 10
```

eg: A regular fun friend for both classes:
```
class B; // forward declaration
class A
{
  int a;
public:
  void set ()
  {
   cout << "\n A a:";
   cin >> a;
  }
  void display ()
  {
   cout <<"\n A a: "<<a;
  }
  friend void swap (A&, B&);
};

class B
{
  int b;
public:
  void set ()
  {
   cout<<" \n B b:";
   cin >> b;
  }
  void display ()
  {
   cout<<"\n B b: " << b;
  }
  friend void swap (A &, B &);
};
```

```
void swap (A &p, B &q)
{
  int r = p.a;
  p.a = q.b;
  q.b = r;
}
void main()
{
  A obja;
  B objb;
  obja.set();
  objb.set();
  obja.display();
  objb.display();
  swap (obja, objb);
  obja.display();
  objb.display();
  return;
}
```

eg: Member function of class A is friend to the class B.

```
class B;
class A
{
  int a;
public:
  void set()
  {
    cout<<"\n A a:";
    cin>>a;
  }
  void display()
  {
    cout<<"\n A a:"<<a;
  }
  void swap (B &);
};
class B
{
  int b;
public:
  void set()
  {
    cout<<"\n B b:";
    cin>>b;
  }
  void display()
  {
    cout<<"\n B b:"<<b;
  }
  friend void A::swap (B &);
};

void A::swap (B &t)
{
  int tmp = a;
  a = t.b;
  t.b = tmp;
}
```

```
void main()
{
 A obja;
 B objb;
 obja.set();
 objb.set();
 obja.display();
 objb.display();
 obja.swap(objb);
 obja.display();
 objb.display();
 return;
}
```

eg:

```
Class A;
Class B
{
 int b;
public:
 void set()
 {
  cout<<"\n B b:";
  cin>>b;
 }
 void display()
 {
  cout<<"\n B b:"<<b;
 }
 void swap(A &);
};

void B::swap(A &t)

Class A
{
 int a;
public:
 void set()
 {
  cout<<"\n A a:";
  cin>>a;
 }
 void display()
 {
  cout<<"\n A a:"<<a;
 friend void B::swap(A &);
}

void B::swap(A &t)
{
 int tmp = t.a;
 t.a = b;
 b = tmp;
}
void main()
{
 A obja;
 B objb;
 obja.set();
```

```
    objb.set();
    obja.display();
    objb.display();
    objb.swap(obja);
    obja.display();
    objb.display();
    return;
}
```

HW:

```
Class Matrix
{
  int x[3][3];
 Public:
  void init();
  void set();
  void display();
  void add(Matrix &, Matrix &);
  void sub(Matrix &, Matrix &);
  void mul(Matrix &, Matrix &);
  void transpose();
};
void Matrix::init()
{
 int i,j;
 for(i=0; i<3; i++)
 {
  for(j=0; j<3; j++)
    x[i][j] = 0;
 }
}
```

```
void Matrix :: set()
{
 int i, j;
 cout << "\n Matrix Data:";
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
    cin >> x[i][j];
 }
}
void Matrix :: display()
{
 cout << "\n Matrix \n");
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   cout << " " << x[i][j];
  cout << "\n";
 }
}
void Matrix :: transpose()
{
 int i=0; j=0; tmp;
 while (i<3)
 {
  for (j=0; j<i; j++)
  {
    tmp = x[i][j];
    x[j][j] = x[j][i];
    x[j][i] = tmp;
  }
  i++;
```

```
void Matrix :: add (Matrix &a, Matrix &b)
{
  int i=0, j=0;
  while (i<3)
  {
    for (j=0; j<3; j++)
      x[i][j] = a.x[i][j] + b.x[i][j];
    i++;
  }
}
```

```
void main ()
{
  Matrix a, b, c;
  a.set ();
  b.set ();
  c.add (a, b);
  a.display ();
  b.display ();
  c.display ();
}
```

- Constructors and Destructors :

i) These are special purpose functions.
ii) Always to be defined in the public section of the class.
iii) These fu^n do not have returning type not even void.
iv) Class name is the function name.
v) Constructor fu^n are used to initialize class objects.
vi) Constructor fu^n are getting called implicitly at the time of object creation.
vii) Constructor fu^n can be called explicitly also.
viii) Constructor fu^n are use to convert basic type data to the user defined type.
ix) Constructor fu^n can be overloaded.
x) There are 3 types of constructor fu^n :
a) Default constructor fu^n : Constructor fu^n with no arguments.

b) Parameterized constructor fu^n : Constructor fu^n with at least one argument.

c) Copy constructor fu^n : Constructor fu^n with same class object as argument.

xi) Destructor fu^n to be prefixed by the sign (~).
xii) Destructor fu^n cannot be overloaded.
xiii) Destructor fu^n cannot be called explicitly. These are getting called automatically when program execution control running out of the scope in which object is created.
xiv) Destructor fu^n are used to release external resources associated with an object.

```
class Number
{
  int no;
public:
 Number ()
 {
  no = 0;
  cout << "\n Default";
 }
 Number (int a)
 {
  no = a;
  cout << "\n Parameterized";
 }
 Number (Number &t)
 {
  no = t.no;
  cout << "\n COPY";
 }
 ~Number ()
 {
  cout << "\n Destructor";
 }
 void setdata ()
 {
  cout << "\n Enter No:";
  cin >> no;
 }
```

```
    int getdata ()
    {
      return no;
    }
    void display ()
    {
      cout<<"\n No:" << no ;
    }
};

void main ()
{
  Number a ;
  a.display ();
  a.setdata ();
  a.display ();
  {
    Number b(10);
    b.display ();
  }
  Number c=15; // only for Turbo C++//
  c.display ();
  c.setdata ();
  c.display ();
  Number d (c); // copy constructor.
d.display ();
  return ;
}
```

- Constructor fun? definations outside the class:

```
# define M 5
class Array
{
  int x[M];
public:
  Array (); //default
  Array (int); // para
  Array (Array &); // copy
  ~ Array () { }
  void setdata ();
  void display ();
  int calctot ();
  int calcavg ()
  {
    return calctot () /M ;
  }
};

Array :: Array ()
{
  for (int i=0 ; i< M ; i++)
    x[i]=0 ;
}

Array :: Array (int p)
{
  for (int i=0 ; i<M ; i++)
    x[i] = p;
}
```

```
Array :: Array (Array &t)
{
 int i=0;
 while (i<M)
 {
  x[i] = t.x[i];
  i++;
 }
}

int Array :: calctot ()
{
 int tot = x[0], i=1;
 while (i<M)
  tot += x[i++];
 return tot;
}

void Array :: setdata ()
{
 int i=0;
 while (i<M)
 {
  cout<<"\n Data:";
  cin >> x[i];
  i++;
 }
}
```

```
void Array :: display ()
{
 cout <<"\n Data : \n";
 for (int i=0; i< M; i++)
   cout <<x[i] <<" ";
 return;
}

void main ()
{
 Array P;   // default
 P.setdata ();
 p.display ();
 Array q(7); // Parameterized
 q.display ();
 q.setdata ();
 q. display ();
 Array r(q); // copy
 r.display ();
 cout <<"\n Total :" <<r.calctot ();
 cout <<"\n Avg :" <<r.calcavg ();
}
```

eg: Class to implement matrix operations.

```
Class Matrix
{
 int x[3][3];
public:
 Matrix ();  // default
 Matrix (Matrix &);  // copy
~Matrix () { }
 void setdata ();
 void display ();
 Matrix add (Matrix &);
 Matrix sub (Matrix &);
 Matrix mul (Matrix &);
 Matrix transpose ();
};

void main()
{
 Matrix P, q, r;
 p.setdata ();
 q.set data ();
 r = p.add (q);
 p.display ();
 q.display ();
 r.display ();
}
```

```
Matrix :: Matrix ()
{
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   x[i][j] = 0;
 }
}

Matrix :: Matrix (Matrix &t)
{
 int i,j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   x[i][j] = t.x[i][j];
 }
}

void Matrix :: setdata ()
{
 int i, j;
 cout << "\n Matrix \n";
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   cin >> x[i][j];
 }
}
```

```
void Matrix :: display ()
{
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
  cout << " " << x[i][j];
  cout << "\n";
 }
}
```

```
Matrix
Matrix :: add (Matrix &a)
{
 Matrix t;
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   t.x[i][j] = x[i][j] + a.x[i][j];
 }
 return t;
}
```

```
Matrix
Matrix :: sub (Matrix &a)
{
 Matrix t;
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   t.x[i][j] = x[i][j] - a.x[i][j];
 }
 return t;
```

```
Matrix
Matrix :: mul (matrix &a)
{
 Matrix t;
 int i=0, j=0, k=0;
 while (i<3)
 {
  for (j=0; j<3; j++)
  {
   for (k=0; k<3; k++)
    t.x[i][j] += (x[i][k] * a.x[k][j]);
  }
  i++;
 }
 return t;
}
```

```
Matrix Matrix :: transpose ()
{
 Matrix t;
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   t.x[i][j] = x[j][i];
 }
 return t;
}
```

source			t		
1	2	3	1	2	3
2	4	6	2	4	6
3	6	9	3	6	9

eg:

```
#define M 5
Class Marks
{
  int rno;
  int mrk[M];
Public:
  Marks();
  ~Marks() { }
  void Getdata();
  void display();
  int total();
  double avg()
  {
    return double(total())/M;
  }
  int getno()
  {
    return rno;
  }
};

Class Result
{
  int no;
  double avg;
Public:
  Result() { no=0;
             avg=0.0;
           }
Result(Marks &t)
{
```


```
  ~Result() { }
  void display()
  {
   cout << "\n Roll No:" << no;
   cout << "\n Avg :" << avg;
  }
   int getno() { return no; }
  double getavg() { return avg; }
};

Marks :: Marks()
{
  rno = 0;
  for(int i=0; i<5; i++)
    mrk[i] = 0;
}

void Marks :: setdata()
{
  cout << "\n Roll No:";
  cin >> rno;
  cout << "\n Marks:";
  for(int i=0; i<M; i++)
    cin >> mrk[i];
}

void Marks :: display()
{
  cout << "\n Roll No:" << rno;
  cout << "\n Marks :\n";
  for(int i=0; i<M; i++)
```

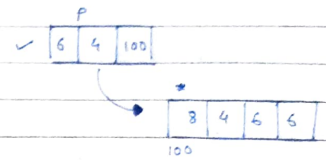
```
cout << "  " << mrk[i];
cout << "\n Total: " << total();
cout << "\n Avg: " << avg();
}

int Marks :: total()
{
 int tot = mrk[0], i=1;
 while (i<M)
 {
  tot+ = mrk[i];
  i++;
 }
 return tot;
}

void main()
{
 Marks m;
 m.setdata();
 m.display();
 Result r(m);
 r.display();
 return;
}
```

eg:

```
class student
{
 int rno;
 char nm[15];
 double mrk;
public:
 student()
 {
  rno = 0;
  nm[] = '\0';
  mrk = 0.0;
 }
 student(int a, char *b, double c=0.0)
 {
  rno = a;
  strcpy(nm, b);
  mrk = c;
 }
 student(student &t)
 {
  rno = t.rno;
  strcpy(nm, t.nm);
  mrk = t.mrk;
 }
 ~student() { }
 void setdata(int a)
 {
  rno = a;
  cout << "\n Name: ";
  cin >> nm;
```

```
cout << "\n Marks : ";
cin >> mrk;
}

void setdata ()
{
 cout << "\n Roll no :";
 cin >> rno;
 setdata (rno);
}
void display()
{
 cout << "\n Roll no : " << rno;
 cout << "\n Name : " << nm;
 cout << "\n Marks : " << mrk;
}
 int getNo () { return rno; }
 double getMarks () { return mrk; }
 Char * getName () { return nm; }
};

void main ()
{
 student s;
 s.setdata (101);
 s.display ();
 student p (6, "Amita", 9.2);
 p.display ();
}
```

eg: Use of pointer as a member of class




```
Class dynobj
{
 int rno;
 int nos;
 int *ptr;
Public :
 dynobj ()
 {
  rno = 0;
  nos = 5;
  ptr = new int [5];
 }
 dynobj (int n)
 {
  rno = 0;
  nos = n;
  ptr = new int [nos];
 }
 dynobj (dynobj &t);
 ~dynobj ()
 {
  delete [] ptr;
 }

 Void setdata ();
 {
 cout << "\n
 void setdata (int);
 void display ();
```

```
int total();
int avg()
{
  return total()/nos;
}
void setMarks();
};

dynobj::dynobj(dynobj &t)
{
  rno = t.rno;
  nos = t.nos;
  ptr = new int [nos];
  for (int i=0; i<nos; i++)
    ptr[i] = t.ptr[i];
}

void dynobj::setdata()
{
  cout<<"\n Roll no:";
  cin>>rno;
  cout<<"\n Marks:";
  for (int i=0; i<nos; i++)
    cin>>ptr[i];
}

void dynobj::setdata(int n)
{
  rno = n;
  cout<<"\n Marks\n";
  for (int i=0; i<nos; i++)
    cin>>ptr[i];
}

int dynobj::total()
{
  int tot = ptr[0], i=1;
  while (i<nos)
    tot += ptr[i++];
  return tot;
}

void dynobj::display()
{
  cout<<"\n Roll no:"<<rno;
  cout<<"\n sub count:"<<nos;
  cout<<"\n Marks:";
  for (int i=0; i<nos; i++)
    cout<<ptr[i]<<" ";
  cout<<"\n Total:"<<total();
  cout<<"\n Avg:"<<avg();
}

void dynobj::SetMarks()
{
  Cout<<"\n Marks\n";
  for (int i=0; i<nos; i++)
    cin>>ptr[i];
}

void main()
{
  dynobj a; // 5 subject
  a.setdata();
  a.display();
  dynobj b(7); // 7 sub
  b.setdata();
  b.display();
```

```
    dynobj c(b); // copy
    c.display();
}
```

eg: Implementation of LinKed List :

Create a header file with new name "NODE.h" and implement class node

```
class Node
{
  int data;
  Node *next;
public:
  Node();
  {
    data=0;
    next = NULL;
  }
  Node (int d)
  {
    data =d;
    next =NULL;
  }
  Node (int d, Node *t)
  {
    data=d;
    next = t;
  }
~Node ()
{ }
int getData ()
{
  return data;
}
Node *getnext ()
{
  return next;
}
void setData (int d)
{
  data = d;
}
void setNext (Node *t)
{
  next = t;
}
  void setNull ()
  {
    next = NULL;
  }
};
```

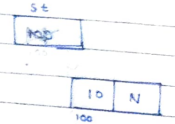
** Close the file.

create a new file and implement class List.

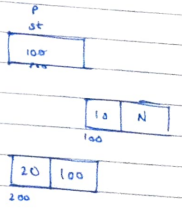
```
#include "NODE.h"
class List
{
  Node *st;
public:
  List ()
```

```
{
  st = NULL;
}
~List();
void addEnd(int d);
void addBegin(int d);
void display();
int getCount();
int getSum();
int getMax();
int getMin();
void delFirst();
void delLast();
void deliNode(int POS);
};

void main()
{
  List p;
  p.addEnd(10);
  p.addEnd(20);
  p.addEnd(30);
  p.addBeg(90);
  p.display();
  cout<<"\n Total Nodes:" << p.getCount();
  cout << "\n Data Sum:" << p.getSum();
  cout <<"\n Min: " << p.getMin();
  cout << "\n Max:" << p.getMax();
  p.delFirst();
  p.display();
}
```

```
void addEnd List :: addEnd(int d)
{
  Node *a = new Node(d);
  if (st == NULL)
    st = a;
  else
  {
    Node *b = st;
    while (b->getNext() != NULL)
      b = b->getNext();
    b->setNext(a);
  }
}
```




```
void addBeg List :: addBeg(int d)
{
  st = new Node(d, st);
}
```




```
void List :: display()
{
  Node *a = st;
  if (st == NULL)
    cout << "Empty list";
  else
    cout << "Data in";
  while (a != NULL)
  {
    cout << a->getData() << " ";
    a = a->getNext();
  }
}
```


```
int List :: getCount ()
{
  int cnt = 0;
  Node *a = st;
  while (a != NULL)
  {
    cnt ++;
    a = a -> getNext ();
  }
  return cnt;
}

int List :: getSum ()
{
  int tot = 0; Node *a = st;
  while (a != NULL)
  {
    tot += a -> getData ();
    a = a -> getNext ();
  }
  return tot;
}

int List :: getMin ()
{
  int min;
  Node *a = st;
  while (a != NULL)
  {
    if (a == st)
      min = a -> getData ();
    else
    {
      if (a -> getData () < min)
        min = a -> getData ();
    }
    a = a -> getNext ();
  }
  return min;
}

int List :: getMax ()
{
  int max;
  Node *a = st;
  while (a != NULL)
  {
    if (a == st)
      max = a -> getData ();
    else
    {
      if (a -> getData () > max)
        max = a -> getData ();
    }
    a = a -> getNext ();
  }
  return max;
}
```

```
Void List :: delFirst ()
{
  Node *a = st;
  if (st == NULL)  //empty list.
    return;
  st = st -> getNext();
  delete a;
}

void List :: delLast ()
{
  if (st == NULL)
    return;
  Node *a=st, *b;
  if (a -> getNext () == NULL)
    st = NULL;
  else
  {
    while (a -> geNext () != NULL)
      b=a;
      a = a -> getNext ();
    }
    b -> setNULL ();
  }
  delete a;
}
```

```
void List :: deliNode (int Pos)
{
  Node *a, *b, *c;
  int i = 1;
  if (st == NULL)
    return;
  if (pos <1)  // invalid request
    return;
  if (pos == 1)
  { delFirst ();
    return;
  }
  a = st;
  while (i<pos && a != NULL)
  {
    b = a;
    a = a -> getNext ();
    i++;
  }
  if (a == NULL)  //no such node
    return;
  if (i == pos) (a -> get next () == NULL)
    delLast ();
  else
  {
    c = a -> getNext ();
    b -> setNext (c);
    delete a;
  }
}
```

```
List :: ~List ()
{
  Node *a = st;
  while (st != NULL)
  {
    st = a -> getNext ();
    delete a;
    a = st;
  }
}
```

eg:

```
Class Point
{
  int x, Y;
public:
  Point ()  // constructor
  {
    x = y = 0;
  }
  Point (int a, int b)
  {
    x = a;
    y = b;
  }
  Point (Point &t)
  {
    x = t.x;  y = t.y.
  }
~Point () {}
```

```
void setData ()
{
  cout << "\n x : ";
  cin >> x;
  cout << "\n y : ";
  cin >> y;
}
void setx (int x)
{
  this -> x = x;
}
```

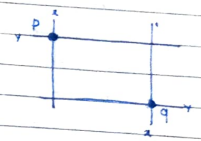
// c++ uses a unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this fu^n was called. The fu^n call a.setx(5); will set the pointer this to the address of object a. The starting address is same as address of the first variable in class structure.

This unique pointer is automatically pass to a member fu^n when it is called. The pointer this acts as an implicit argument to all member fu^n.

```
void sety (int y)
{
  this -> Y = Y;
}
void setData (int x, int y)
{
  this -> x = x;
  this -> y = y;
}
```

```
int getx() { return x; }
int gety() { return y; }

void display()
{
 cout << "\n x : " << x;
 cout << "\n y : " << y;
}
};

Class Rect
{
 Point p;
 Point q;
Public:
 Rect()
 {
  p = Point(); // explicit call
  q = Point();
 }
 Rect(int a, int b, int c, int d)
 {
  p = Point(a,b);
  q = Point(c,d);
 }
 Rect(Point a, Point b)
 {
  p = a;
  q = b;
 }
 ~Rect() {}
```

```
void setRect(int a, int b, int c, int d)
{
 p.setData(a,b);
 q.setData(c,d);
}
void display()
{
 p.display();
 q.display();
}
int getwidth()
{
 return q.getx() - p.getx();
}
int getHeight()
{
 return q.gety() - p.gety();
}
int area()
{
 return getwidth() * getHeight();
}
};
void main()
{
 Rect r(10,10,30,30);
 r.display();
 cout << "\n Area : " << r.area();
 Rect a;
 a.setRect(100,100,200,200)
 a.display();
```



```
cout<<"In width :"<<a.getwidth();
cout<<" In Height :"<<a.getHeight();
}
```

- **Operator Overloading:**

Operator overloading is one of the many exiting features of C++ language. It is an important technique that has enhansed the power of extensibility of C++. As we know C++ tries to make the user defined data types behave in much the same way as the built in types. For instance C++ permits us to add two variables of user defined types with the same syntax ie applied to the data type.

- ~~The mechan~~ This means that C++ has the ability to provide the operators with a special meaning for the datatype. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides a flexible option for creation of new definations for the most of the C++ operators. we can overload all the C++ operators except the following:

1) Class member access operators (.,.*)
2) Scope resolution operator (::)
3) Sizeof operator
4) Conditional operators.

The semantics of operator can be extended, we cannot change the syntax, the grammatical rules that governs as use such as no. of operands, precedence and associativity.

Defining Operator Overloading:

To define an additional task to an operator we must specify what it means in relation to the class to which the operator is applied. This is done with the special fu? called operator fu? which describes the task. following is the general format for operator function:

```
< return type> <class name> :: operator <~~OP~~ OP> (argument)
{
    ——— ;
    ——— ;
    return (return value);
}
```

where return type is the type of value return by the special operation and 'OP' is the operator being overloaded. 'Operator' 'OP' is a fu? name where operator is keyword.

Operator fu? must be either member fu? or friend fu? the basic diff. b/w them is that friend fu? will have only one argument for unary operators and two for binary operators. ~~This is because~~ while a member fu? has no arguments for unary operators and only one for binary operator. This is because the object used to invoke member function

is passed implicitly and therefore is available for member function.

This is not the case with friend fn. The argument can be passed by value or reference

```
Class Result
{
 int no;
 double mrk;
public:
  Result () // default
  {
   no=0;
   mrk=0.0;
  }
  Result (Result &t)  // copy
  {
   no = t.no;
   mrk = t.mrk;
  }
 ~Result(){ }
 void setData ()
 {
  cout << " Roll NO:";
  cin >> no;
  cout << "\n Mark:";
  cin>> mrk;
 }
 void display ()
 {
  cout << "\n Roll no:" << no;
  cout << "\n Marks:" << mrk;
 }
 int operator == (int n)
 {
   return (no==n);
 }
 int operator > (int n)
 {
   return (no>n);
 }
};

void main()
{
  Result a;
  a.setData();
  a.display();
  if (a>6)
    cout << "\n found";
  else
    cout << "\n Not found";
 }
// continue from display()
```

```
int operator < (int n)
{
return (no<n);
}
int operator > (double m)
{
return (mrk>m);
}
int operator< (double m);
{
return (mrk<m);
}
int operator > (Result &t)
{
return (no>t.no);
}
int operator < (Result &t)
{
return (no< t.no);
}
int operator == (Result &t)
{
return (no== t.no);
}
Result operator + (double a)
{
Result t;
t.no = no;
t.mrk = mrk +a ;
return t;
}
```

```
Result operator - (double a)
{
Result t;
t.no = no;
t.mrk = mrk - a ;
return t;
}
}; // store the above contents with name "result.h".
void main ()
```

```
#include "result.h"
void main ()
{
Result r;
r.setData ();
r.display ();
Result t = r+1.2; //(only for turbo c++)
                  for other editor
                  Result t;
                  t = r+1.2;
t.display();
t.SetData ();
if (r>t)
  r.display ();
else
  t.display ();
}
```

// Array of ~~type~~ object of type result.

```
#include "result.h"
void main ()
{
  Result p[5];
  int i=0;
  while (i<5)
  {
    p[i].setData ();
    i++;
  }
  for (i=0; i<5; i++)
    p[i].display ();
}
```

// sorting of objects of type result.

```
#include "result.h"
void main ()
{
  Result p[5], tmp;
  int i=0, j=0, pos;
  while (i<5)
  {
    p[i].setData ();
    i++;
  }
  cout<<"\n Before Sorting \n";
  for (i=0; i<5; i++)
    p[i].display ();
  for (i=0; i<4; i++)
  {
    pos = i;
    for (j=i+1; j<5; j++)
    {
      if (p[j] < p[pos])
        pos = j;
    }
    if (i!= pos)
    {
      tmp = p[i];
      p[i] = p[pos];
      p[pos] = tmp;
    }
  }
  cout<<"\n After Sorting \n";
  for (i=0; i<5; i++)
    r[i].display ();
}
```

eg:

```
class Number
{
  int no;
public:
  Number()
  {
    no=0;
  }
  Number(int n)
  {
    no=n;
  }
  Number(Number &t)
  {
    no=t.no;
  }
  ~Number(){}
friend istream & operator >>(istream &in, Number &t)
{
  cout<<"\n Enter No:";
  in>>t.no;
  return in;
}
friend ostream & operator <<(ostream &out, Number &t)
{
  out<<"\n No:"<<t.no;
  return out;
}
```

```
Number operator +(int n)
{
  Number t(no+n);
  return t;
}
Number operator -(int n)
{
  Number t(no-n);
  return t;
}
friend Number operator +(int n, Number &t)
{
  Number K(n+t.no);
  return K;
}
friend Number operator -(int n, Number &t)
{
  Number K(n-t.no);
  return K;
}

int operator >(Number &t)
{
  return(no>t.no);
}
int operator >(int n)
{
  return(no>n);
}
```

```
int operator <(Number &t)
{
  return (no < t.no);
}
int operator <(int n)
{
  return (no < n);
}
int operator == (Number &t)
{
  return (no == t.no);
}
int operator ==(int n)
{
  return (no == n);
}
```

Main program

```
void main()
{
  Number p,q;
  cin>>p;
  cout<<p;
  q = p+2;
  cout<<q;
  q = 5+p;
  cout<<q;
  if (p>q)
    cout<<p;
  else
    cout<<q;
}
```

Continuation:

```
Number operator += (int a)
{
  no = no + a;
  return *this;
}
Number operator -= (int a)
{
  no = no - a;
  return *this;
}
Number operator ++ () // preincrement
{
  no = no+1;
  return *this;
}
Number operator ++ (int) // postincrement
{
  Number t(no);
  no = no+1;
  return t;
}
};

void main()
{
  Number P(7), q;
  p+=3;
  cout<<p;
  q = ++p;
  cout<<p<<q;
  q = p++;
  cout<<p<<q;
}
```

eg:

```
class Matrix
{
 int x[3][3];
public:
 Matrix();
 Matrix (Matrix &);
 ~Matrix() {}
 friend istream & operator >> (istream &, Matrix &);
 friend ostream & operator << (ostream &, Matrix &);
 Matrix operators + (Matrix &);
 Matrix operators - (Matrix &);
 Matrix operators * (Matrix &);
 void operator - ();
};

void main()
{
 Matrix p,q,r;
 cin >> p;
 cin >> q;
 r = p+q;
 cout << p << q << r;
 -r;
 cout << r;
}
```

```
Matrix :: Matrix ()
{
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   x[i][j] = 0;
 }
}

Matrix :: Matrix (Matrix &p)
{
 int i, j;
 for (i=0; j<3; i++)
 {
  for (j=0; j<3; j++)
   x[i][j] = p.x[i][j];
 }
}

istream & operator >> (istream &in, Matrix &p)
{
 int i, j;
 cout'<<"\n Matrix Data \n");
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   in >> p.x[i][j];
 }
 return in;
}
```

eg:

```
class Matrix
{
 int x[3][3];
public:
 Matrix();
 Matrix (Matrix &);
 ~Matrix() {}
 friend istream & operator >> (istream &, Matrix &);
 friend ostream & operator << (ostream &, Matrix &);
 Matrix operators + (Matrix &);
 Matrix operators - (Matrix &);
 Matrix operators * (Matrix &);
 void operator - ();
};

void main()
{
 Matrix p,q,r;
 cin>>p;
 cin>>q;
 r = p+q;
 cout<<p<<q<<r;
 r-;
 cout<<r;
}
```

```
Matrix :: Matrix ()
{
 int i,j;
 for(i=0; i<3; i++)
 {
  for(j=0; j<3; j++)
   x[i][j]=0;
 }
}

Matrix :: Matrix (Matrix &p)
{
 int i,j;
 for(i=0; j<3; i++)
 {
  for(j=0; j<3; j++)
   x[i][j] = p.x[i][j];
 }
}

istream & operator >> (istream &in, Matrix &p)
{
 int i,j;
 cout<<"\n Matrix Data \n");
 for(i=0; i<3; i++)
 {
  for(j=0; j<3; j++)
   in >> p.x[i][j];
 }
 return in;
}
```

```
ostream & operator << (ostream & out , Matrix & p)
{
 out << "\n Matrix \n";
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   out << " " << p.x[i][j];
   out << "\n";
 }
 return out;
}

~~Matrix : Matrix operator + (Matrix &~~

Matrix Matrix :: operator + (Matrix &q)
{
 Matrix t;
 int i, j;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   t.x[i][j] = x[i][j] + q.x[i][j];
 }
 return t;
}
Matrix Matrix :: operator - (Matrix &q)
{
Matrix t;
int i, j;
```

```
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
   t.x[i][j] = x[i][j] - q.x[i][j];
 }
 return t;
}

Matrix Matrix :: operator * (Matrix &q)
{
 Matrix t;
 int i, j, K;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
  {
   for (K=0; K<3; K++)
    t.x[i][j] += (x[i][K] * q.x[K][j]);
  }
 }
 return t;
}

void Matrix :: operator - ()
{
 int i=0, j=0, tmp;
 for (i=0; i<3; i++)
 {
  for (j=0; j<3; j++)
  {
   tmp = x[i][j];
   x[j][j] = x[j][i];
   x[j][i] = tmp;
```

- Type Conversion :

Source	Destination
1. Basic	User Defined (By defining constructor)
2. User defined by defining operator <destination>()	Basic
3. User defined @ By defining fu": operator <desti>()	User Defined ⓑ By defining constructor fu" with source as argument.

- Characteristics of operator () :

i) This are special purpose fu"
ii) Always to be defined in the public section of the class.
iii) This functions do not have returning type not even void.
iv) This fu" do not have arguments hence we cannot overload this fu"
v) This fu" do not have returning type not even void but returning function is must.
vi) fu" name is the returning type of fu".
vii) The general format for the function is

Syntax:

```
operator <type> ()
{
  ------;
  ------;
  return <ret-value>;
}
```

- Type conversion from user defined data type to basic data type:

```
#define M 5
class Array
{
 int x[M];
public:
 Array ();
 ~Array () { }
friend istream & operator >> (istream &, Array &);
friend ostream & operator << (ostream &, Array &);
int total ();
operator int ()
{
 return tot;
}
operator double ()
{
int tot = *this ; // type conversion from Array to int.
return double (tot)/M ;
}
};
```

```
void main()
{
Array p;
cin>>p;
cout<<p;
int tot = p;
double avg = p;
Cout <<"\n Total :"<<tot;
Cout<<"\n Avg :"<<avg;
return;
}

Array :: Array ()
{
for(int i=0; i<M; i++)
 {
  x[i] = 0;
 }
}

istream & operator >> (istream &in, Array &p)
{ cout<<"Array data:";
 for (int i=0; i<M; i++)
 {
  in >> p.x[i];
 }
return in;
}
```

```
ostream & operator << (ostream &out, Array &p)
{
 for (int i=0; i<M; i++)
 {
  out <<" " <<p.x[i];
 }
 out <<"\n";
 return out;
}

int Array :: total ()
{
 int tot = x[0], i=1;
 while (i<M)
  tot += x[i++];
 return tot;
}
```

- Class to class type Conversion :

```
class Marks
{
 int no;
 int mrk[5];
public:
 Marks ();
 ~Marks () { }
 friend istream & operator >>
   (istream &, Marks &);
 friend ostream & operator <<
   (ostream &, Marks &);
```

```
int total();
operator int ()
{
return no;
}
operator double ()
{
int tot = total();
return double(tot)/5;
}

class Result
{
int no;
double mrk;
public:
Result () { no=0; mrk=0.0; }
Result (Marks &t)
{
no=t;  // conversion from marks
mrk=t;     to basic type
}
~Result () { }
friend ostream & operator << (ostream &out, Result &t
{
out <<"\n Roll No: " << t.no <<"\n Marks: " << t.mrk
return out;
}
};
```

```
Marks :: Marks()
{
no;
for (int i=0; i<5; i++)
mrk[i]=0;
}

int Marks :: total()
{
int tot = mrk[0], i=1;
for (; i<5; i++)
tot += mrk[i];
return tot;
}
istream & operator >> (istream & in, Marks & t)
{
cout <<"\n Roll No: "; in >> t.no;
cout <<"\n Marks:";
for (int i=0; i<5; i++)
in>> t.mrk[i];
return in;
}

ostream & operator << (ostream &out, Marks &t)
{
out <<"\n Roll no: " << t.no;
out <<"\n Marks:";
for (int i=0; i<5; i++)
out << t.mrk[i] <<" ";
return out;
}
```

```
void main()
{
Marks m;
cin>> m;
cout << m;
Result r(m);
cout << r;
}
```

eg:

```
class Result
{
 int no;
 double mrk;
public:
 Result()
 {
  no=0;
  mrk=0.0;
 }
 Result(int a, double b)
 {
  no=a;
  mrk=b;
 }
 Result(Result &t)
 {
  no=t.no;
  mrk=t.mrk;
 }
 ~Result(){}
 friend ostream & operator << (ostream &out,
                               Result &t)
 {
  out <<"\n Roll no: "<< r.no << "\n Marks :"
      << r.mrk << "\n";
  return out;
 }
};

class Marks
{
 int no;
 int mrk;
public:
 Marks();
 ~Marks(){}
 int total();
 operator int()
 {
  return no;
 }
 operator double()
 {
  int tot = total();
  return double(tot)/5;
 }
 operator Result()
 {
  int b = *this;  // type conv. from marks to int
  double c = *this; // type conv. from marks to double
  Result a(b,c);
  return a;
 }
```

```
friend istream &operator >> (istream&, Marks &);
friend ostream &operator << (ostream &, Marks &);
};

istream &operator >> (istream &in, Marks &r)
{
cout << "In Roll no: ";
in >> r.no;
cout << "In Marks :";
for (int i=0; i<5; i++)
   in >> r.mrk[i];
return in;
}

ostream &operator << (ostream &out, Marks &r)
{
out << "In Roll No :" << r.no;
out << "In Marks: ";
for (int i=0; i<5; i++)
  out << " " << r.mrk[i];
return out;
}
int Marks :: total ()
{
int tot = mrk[0], i=1;
while (i<5)
{
 tot += mrk[i++];
}
return tot;
}
```

```
Marks :: Marks ()
{
no = 0;
for (int i=0; i<5; i++)
   mrk[i] = 0;
}

void main ()
{
marks m;
cin >> m;
cout << m;
Result r;
r = m;
cout << r;
return;
}
```

Inheritance :

* Types of inheritance :

1) Single :

A - base class
↓
B - derived class

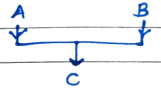
A is parent class of B

2) Multilevel :

A - base
↓
B - derived / base
↓
C - derived

A is parent class of B & B is parent class of C
* but A is not parent class of C.

3) Multiple :

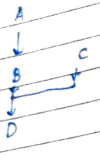


Syntax:

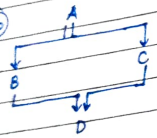
```
class A1
{
};
class A2
{
};
class B : public A1, public A2
{
};
```

4) Hybrid :

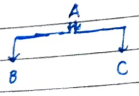
ⓐ



ⓑ



5) Hierarchical :



syntax:

```
class A
{
};
class B1 : public A
{
};
class B2 : public A
{
};
```

- Types of class derivation :

i) Private - (by default)
ii) public

- Syntax for class derivation :

class <derived class> : [<derivation type>] <list of base class>

```
{
  ____ ;
  ____ ;
  ____ ;
};
```

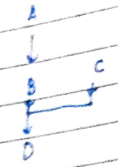
Base class	Declaration type	Derived class
private	private	NA
private	public	NA
protected	private	private
protected	public	protected
public	private	private
public	public	public

1) // program to implement single inheritance with public derivation

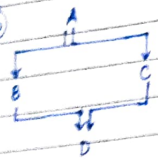
```
class Number
{
 int no;
public:
  void setData ();
  {
```

4) Hybrid :

(a)



(b)



5) Hierarchical :

syntax:

```
class A
{
};
class B1 : public A
{
};
class B2 : public A
{
};
```

- Types of class derivation:

i) private - (by default)
ii) public

- Syntax for class derivation :

```
class <derived class> : [<derivation type>] <list of base class>
{
    ____ ;
    ____ ;
    ____ ;
};
```

Base class	Declaration type	Derived class
private	private	NA
private	public	NA
protected	private	private
protected	public	protected
public	private	private
public	public	public

1) // program to implement single inheritance with public derivation

```
class Number
{
 int no;
public:
  void setData();
{
```

```
    cout << "\n Enter NO: ";
    cin >> no;
}
void display ()
{
    cout << "\n NO: " << no;
}
int getNo ()
{
    return no;
}
};
```

Number

private :
int no
public :
void setData();
void display();
int getNo();

```
class ExNumber : public Number
{
public :
    void table ();
    void prime ();
};
```

ExNumber

public :
void setData();
void display();
int getNo();
void table();
void prime();

```
void ExNumber :: table ()
{
    int i = 1, n = getNo ();
    cout << " Table\n";
    for (int i=1; i<=10; i++)
        cout << " " << n * i ;
}

void ExNumber :: prime()
{
    int i = 2, n = getNo ();
    while (i<n)
    {
        if (n % i == 0)
            break;
        i++;
    }
    if (i == n)
        cout << "\n Prime No.";
    else
        cout << "\n Not prime no";
}

void main ()
{
    Number p;
    p.setData ();
    p.display ();
    ExNumber q;
    q.setData ();
    q.display ();
    q.table ();
}
```

- Single inheritance with private derivation.

```
class number
{
  int no;
public:
  void setData()
  {
   cout<<"Enter NO: ";
   cin>>no
  }
  void display()
  {
   cout<<"no: "<<no;
  }
  int getNo()
  {
   return no;
  }
};

class ExNumber : private Number
{
  public:
    void prime();
    void table();
    void input()
    {
     setData();
    }
```

ExNumber

private:
- void setData()
- void display()
- int getNo()

public:
- void input()
- void output()
- void table()
- void prime()

```
    void output()
    {
     display();
    }
};

void ExNumber::table()
{
  int i=1, n=getNo();
  cout<<"n Table n"
  for(int i=1; i<=10; i++)
    cout<<" "<<n*i;
}

void ExNumber :: prime()
{
  int i=2, n=getNo();
  while(i<n)
  {
   if(n%i==0)
     break;
   i++;
  }
  if(i==n)
  cout<<"Prime NO";
  else
   cout<<"Not prime";
}
```

```
void main()
{
  Number p;
  p.setData();
  p.display();
  Exnumber q;
  q.input();
  q.output();
  q.table();
}
```

- Multilevel Inheritance :

i) use of multilevel inheritance.
ii) use of constructors
iii) use of protected members

```
class Number
{
 protected:
   int no;
 public:
   Number()
   {
    no = 0;
   }
   Number(int n)
   {
    no = n;
   }
   ~Number() { }
   void setNumber()
   {
    cout << "\n No:";
    cin >> no;
   }
   void display Number()
   {
    cout << "\n No:" << no;
   }
};

class student : public Number
{
 protected:
   char nm[15];
 public:
   student(): Number()
   {
    nm[0] = '\0';
   }
   student(int a, char *b) : Number(a)
   {
    strcpy(nm, b);
   }
   ~student() { }
   void setName()
   {
    cout << "\n Name:";
    cin >> nm;
   }
```

```
void displayName()
{
cout << "Name : " << nm;
}
};

class Marks : public student
{
protected:
double mrk;
public:
Marks() : student()
{
mrk = 0.0;
}
Marks(int a, char *b, double c) : student(a,b)
{
mrk = c;
}
~Marks() {}
void setMarks()
{
cout << "In Marks:";
cin >> mrk;
}
void displayMarks()
{
cout << "\n Marks:" << mrk;
}
};
```

```
void main()
{
Number a(10);
a.displayNumber();
student b;
b.setNumber();
b.setName();
b.displayNumber();
b.displayName();
Marks c(12, "Ramesh", 87.92);
c.displayNumber();
c.displayName();
c.displayMarks();
}
```

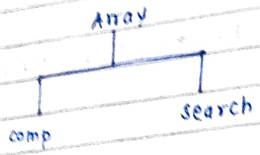
eg:

```
class Number
{
protected:
int no;
public:
Number()
{
no = 0;
}
Number(int n)
{
no = n;
}
```

```
~Number () { }
void setNumber ()
{
  cout << "\nNO:";
  cin >> no;
}
void display ()
{
  cout << "NO: " << no;
}
};

class student : public Number
{
protected :
  char nm[15];
public:
  student (): Number ()
  {
    nm[0] = '\0';
  }
  student (int a, char *b) : Number (a)
  {
    strcpy (nm, b);
  }
  ~student () { }
  void setData ()
  {
    Number:: setData ();
    cout << "\n Name :";
    cin >> nm;
  }
  void display ()
  {
    Number :: display ();
    cout << "\n Name : " << nm;
  }
};

class Marks : public student
{
protected :
  double mrk;
public :
  Marks () : student ()
  {
    mrk = 0.0;
  }
  Marks ( int a , char *b, double c): student(a, b)
  {
    mrk = c;
  }
  ~ Marks () { }
  void set Data ()
  {
    Student :: setData ();
    Cout << " \n Name : ";
    cin >> nm ;
  }
  void display ()
  {
    Student :: display ();
    Cout << " Marks: " << mrk;
  }
```

```
void main()
{
  Number n[10];
  n.display();
  student b;
  b.setData();
  b.display();
  Marks c;
  c.setData();
  c.display();
}
```

- Program to implement Hierarchical inheritance.




```
#define N 5
class Array
{
protected:
  int x[N];
public:
  Array();
  ~Array(){}
  void setData();
  void display();
};
```

```
Array :: Array()
{
  for(int i=0; i<M; i++)
    x[i] = 0;
}

void setData();
void Array :: setData()
{
  int i=0;
  while(i<M)
  {
    cout<<"\n Data:";
    cin>>x[i];
    i++;
  }
}

void Array :: display()
{
  cout<<"\n Data:";
  for(int i=0; i<M; i++)
    cout<<x[i];
}
```

```
class comp : public Array
{
  public :
    comp () : Array ()
    {}
    ~comp () {}
    int total ();
    int avg ();
    int max ();
    int min ();
};

int comp :: total ()
{
  int tot = x[0], i=1 ;
  while (i<M)
    tot += x[i++];
  return tot;
}

int comp :: avg ()
{
  return total ()/M ;
}

int comp :: max ()
{
  int mx = x[0] , i=1;
  while (i<M)
  {
    if (x[i] > mx)
      mx = x[i] ;
    i++;
  }
  return mx ;
}

int comp :: min ()
{
  int mn = x[0] , i=1;
  while (i<M)
  {
    if (x[i] < mn)
      mn = x[i];
    i++;
  }
  return mn ;
}

class search : public Array
{ public
  protected :
    search () : Array ()
    {}
    ~search () {}
    int ssearch (int);
    int bsearch (int);
    int isordered ();
};
```

```
int search :: ssearch (int sv)
{
  int i=0;
  while (i<M)
  {
    if (x[i]==sv)
      break;
    i++;
  }
  return (i!=M);
}

int search :: isOrdered ()
{
  int i=1;
  while (i<M)
  {
    if (x[i] < x[i-1])
      break;
    i++;
  }
  return (i==M);
}

int search :: bsearch (int sv)
{
  int l=0, r=m-1, mid;
  while (l<=r)
  {
    mid = (l+r)/2;
    if (x[mid]==sv)
      break;
    if (sv<x[mid])
      r = mid-1;
    else
      r = mid+1;
  }
  if (l>r)
    return 0;
  else
    return 1;
}

void main ()
{
  comp p;
  p.setData ();
  p.display ();
  cout << "\n Total : " << p.total () << "\n Avg : " << p.avg();
  search K;
  K.setData ();
  K.display ();
  int res;
  if (K.isOrdered ())
    res : K.bsearch (13);
  else
    res : K.Ssearch (13);
  if (res ==0)
    cout << "\n Not Found ";
  else
    cout << "\n Found";
}
```

- Implementation of multiple inheritance


```
class Theory
{
 protected
   int mrk[6];
 public:
   Theory ();
   ~Theory () {}
   void setData ();
   void display ();
   int total ();

};

class Practical
{
 protected:
   int p1, p2;
 public:
   Practical () { p1 = p2 = 0; }
   ~Practical () {}
   void setData ();
   void display ();
   int total ();
};
```

```
class Student : public Theory, public Practical
{
 protected:
   int no;
   char nm[15];
 public:
   Student () : Theory (), Practical ()
   {
     no = 0;
     nm[0] = '\0';
   }
   ~Student () { }
   void setData ();
   void display ();
   void displayResult ();
   {
     Student :: display ();
     Theory :: display ();
     Practical :: display ();
   }
};

Theory :: Theory ()
{
  int i = 0;
  while (i < 6)
  { mrk[i] = 0;
    i++
  }
}
```

```
void Theory :: setData ()
{
  int i=0;
  while(i<5)
  {
  cout << "Marks:";
  cin >> x[i];
  i++;
  }
}

void Theory :: display ()
{ cout << "Marks:";
for(int i=0; i<5; i++)
  {
  cout << " " << x[i];
  }
}

int Theory :: total()
{
  int tot = x[0], i=1;
  while (i<5)
  {
  tot += x[i++];
  return tot;
  }
}
```

```
void Practical :: setData()
{
  cout << "Practical sub 1:";
  cin >> P1;
  cout << "\n Practical sub 2";
  cin >> P2;
}

void Practical :: display()
{
  cout << "Practical Marks:" << "\n P1:" << P1 <<
       "\n P2:" << P2;
}

int Practical :: total()
{
  return P1+P2;
}

void student :: setData()
{
  cout << "\n Roll No:";
  cin >> no;
  cout << "\n Name:";
  cin >> nm;
  Theory :: setData();
  Practical :: setData();
}
```

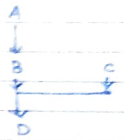
```
void Theory :: setData ()
{
  int i=0;
  while(i<5)
  {
  cout << "Marks:";
  cin >> x[i];
  i++;
  }
}

void Theory :: display ()
{ cout << "Marks:";
for(int i=0; i<5; i++)
  {
   cout << " " << x[i];
  }
}

int Theory :: total()
{
  int tot = x[0], i=1;
  while (i<5)
  {
   tot += x[i++];
  return tot;
  }
}
```

```
void Practical :: setData()
{
  cout << "Practical Sub 1:";
  cin >> P1;
  cout << "\n Practical sub2";
  cin >> P2;
}

void Practical :: display()
{
  cout << "Practical Marks:" << P1: " << P1 <<
       "\n P2:" << P2;
}

int Practical :: total ()
{
  return P1+P2;
}

void student :: setData()
{
cout << "\n Roll No: ";
cin >> no;
cout << "\n Name:";
cin >> nm;
Theory :: setData();
Practical :: setData();
}
```

```
void Student :: display()
{
cout<<"\n Roll No: "<<no<<"\n Name :"<<nm;
Theory :: display();
Practical
}

void main()
{
Student s;
s.setData();
s.displayResult();
cout<<"\n Theory Total:";
cout<<s.Theory :: total();
cout<<"\n Practical Total:";
cout<< s.Practical :: total();
}
```

```
void Student :: setData()
{
```

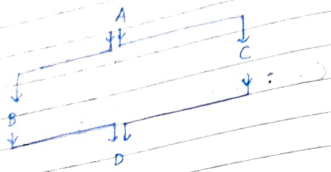
implementation of Hybrid inheritance :




```
class Number
{
  protected:
    int no;
  public
    Number()
    {
      no=0;
    }
   ~Number{}
    void setData()
    {
     cout<<"\n NO:";
     cin>>no;
    }
    void display()
    {
     cout<<"\nNO:"<<no;
    }
};

class Name : public Number
{
 protected:
    char nm[15];
 public:
```

```
Name() : Number()
{
 nm[0] = '\0';
}
~Name() { }
void setData()
{
 Number :: setData();
 cout << "\n Name :";
 cin >> nm;
}
void display()
{
 Number :: display();
 cout << " Name : " << nm;
}
};

class Sports
{
protected:
 int smrk;
public:
 Sports()
 {
  smrk = 0;
 }
 ~Sports() { }
 void setData()
{
 cout << "\n Sports Marks :";
 cin >> smrk;
}
void display()
{
 cout << "\n Sports Marks :";
 cout << smrk;
}
};

class Result : public Name, public Sports
{
protected :
 int tmrk;
public :
 Result() : Name(), Sports()
 {
  tmrk = 0;
 }
 ~Result() { }
 void setData()
 {
  Name :: setData();
  Sports :: setData();
  cout << " Marks : ";
  cin >> tmrk;
 }
 void display()
 {
  Name :: display();
  Sports :: display();
  cout << "\n Theory : " << tmrk;
 }
};
```

```
void main()
{
  result r;
  r.setdata();
  r.display();
}
```




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

consider a situation where all the three kinds of inheritances namely multilevel, multiple and hierarchical are involved as shown in above diagram the class D has two direct base classes B and C which themselves have a common base class A. The D inherits the traits of A via two separate paths it can also inherit directly as shown by the broken lines. The class A is sometimes referred to as indirect base class. The inheritance by the class B as shown in above diagram might pose some problems all the public and protected member of A are inherited into D twice 1st via B and again via C. This means the D would have duplicate sets of members inherited from A. This introduces ambiguity and should be resolved.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class. while declaring the direct or intermediate base classes as shown below.

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

```
class D : public B , public C
{
};
```

When a class is made virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inherit paths exist b/w the virtual base class and deriv class.

- for overriding :

```
class A
{
  int a;
public:
  void set()
  {
    cout << "\n A a : ";
    cin >> a;
  }
  void display()
  {
    cout << "\n A a : " << a;
  }
};

class B : public A
{
  int b;
public:
  void set()
  {
    cout << "\n B b : ";
    cin >> b;
  }
  void display()
  {
    cout << "\n B b : " << b;
  }
};

class C : public B
{
  int c;
public:
  void set()
  {
    cout << "\n C c : ";
    cin >> c;
  }
  void display()
  {
    cout << "\n C c : " << c;
  }
};

void main()
{
  A obja;
  obja.set();
  obja.display();
```

```
B objb;
objb.A::set();
objb.set();
objb.A::display();
objb.display();
C objc;
objc.set();
objc.display();
}
```

- pointer to object :

```
void main()
{
A obja;
A *ptr;
ptr = &obja;
ptr -> set();
ptr -> display();
}
```

Here above program 'obja' is an object of class type 'A' similarly we had pointer ptr of class type A then pointer ptr is initialized with the address of obja. we can refer to the member fun^n of class A in two ways, one by using (.) dot operator and another by using (->) arrow operator. So, the statements

```
obja.set();
obja.display();
```

are equivalent to

```
ptr -> set();
ptr -> display();
```

since *ptr is an alias for obja so we can also use,

```
(*ptr).set();
(*ptr).display();
```

The parenthesis is necessary because (.) dot operator is having higher precedence than the indirection operator (*).

We can also create objects using pointers and new operator as follows:

```
A *ptr = new A;
ptr -> set();
ptr -> display();
```

This statement allocates enough memory for the data members in the object structure and assigns the object of memory space to ptr. Now, ptr can be used to refer to the members as shown above.

- Pointers to derived classes :

```
void main()
{
A obja;
A *aptr = &obja;
aptr -> set();
aptr -> display();
B objb;
aptr = &objb;
aptr -> set();
aptr -> display();
```

```
((B*)aptr) -> set();
((B*)aptr) -> display();
C *cptr;
C objC;
cptr = &objC;
cptr -> set();
cptr -> display();
}
```

Consider Multilevel inheritance is as shown in above dig. we can use pointers not only to the base objects but also to the objects of derived classes. Pointers to object of base class are type compatible with pointers of objects of derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes.

eg: if A is a base class and B is derived class from A then pointer declare as pointer to A can also be a pointer to B.

eg:

```
A *aptr;  // pointer to class A
A objA;   // base class object
B objB;   // object of derived class
aptr = &objA; // aptr points to objA.
```

We can make aptr to point to the object as follows `aptr = &(objB)` // aptr pts to objb this is perfectly valid with C++ because objb is a object derived from class A.

However, there is problem in using aptr to access public members of derived class B. using aptr we can access only those members which are inherited from A and not the members that originally belong to B. Incase, a member of B has the same name as one of the member of A, then any reference to that members by aptr will always access the base class members.

Although C++ permits a base pointer to point to any object that derived from that base the pointer cannot be directly use to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

• Virtual Functions:

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message but in diff. from an essential requirement of polymorphism is the ability to refer two objects without any regard of their classes. this nessessiates a use of single ptr variable to refer to objects of diff. classes.

```
class A
{
  int a;
public:
 virtual void set()
 {
  cout << "\n A a:";
  cin >> a;
 }
 virtual void display()
 {
  cout << "\n A a:" << a;
 }
};

class B : public A
{
  int b;
public:
 void set()
 {
  cout << "\n B b:";
  cin >> b;
 }
 void display()
 {
 cout << "\n B b:" << b;
 }
};
```

```
class C : public B
{
  int c;
public:
  void set()
  {
   cout << "\n C c:";
   cin >> c;
  }
  void display()
  {
   cout << "\n C c:" << c;
  }
};

void main()
{
  A obja;
  obja.set();
  obja.display();
  B objb;
  objb.A::set();
  objb.set();
  objb.A::display();
  objb.display();
  C objc;
  objc.set();
  objc.display();
}
```

```
void main()
{
A *aptr;
A obja;
aptr = &obja;
aptr -> set();
aptr -> display();
B objb;
aptr = &objb;
aptr -> set();
aptr -> display();
C objc;
aptr = &objc;
aptr -> set();
aptr -> display();
}
```

In the above program we had used a pointer to base class to refer to all the derived objects. but here before we observed that a base pointer even when it is made to contain address of a derived class, always execute fu^n in the base class. The compiler simply ignores contents of the pointer and choose the member fu^n that matches the type of the pointer. Then que. arises how to achieve runtime polymorphism. it is achieved using what is known as virtual fu^n.

When we use the same fu^n name in both base and derived classes, the fu^n in base class is declared as virtual using the keyword virtual preceding is normal declaration. when a fu^n is made virtual, C++ determines which fu^n to use at runtime based on the type of object pointed to by the base pointer, rather than the type of pointer. Thus, by making the base pointer to point to diff. objects we can execute diff. versions of the virtual fu^n as shown in the above program.

One imp. point to remember is that we must access virtual fu^n through the use of a pointer declared as a pointer to the base class. the runtime polymorphism is achieved only when a virtual fu^n is accessed through a pointer to the base class.

- Rules for virtual fu^n

① The virtual fu^n must be the members of some class.
② They cannot be static members.
③ They are accessed by using object pointers.
④ A virtual fu^n can be a friend of another class.
⑤ A virtual fu^n in a base class must be defined even though it may not be used.
⑥ The prototype of base class version of a virtual fu^n and all the derived class versions must be identical. if two fu^n with the same name have diff. prototypes. C++ considered them as overloaded fu^n and the virtual fu^n

mechanism is ignored

② we cannot have virtual constructors but we can have virtual destructors.

③ while a base pointer can point to any type of derived object the reverse is not true. that is we cannot use pointer to derived class to access object of base type.

④ when a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. it is incremented or decremented only related to base type, therefore we should not use this method to move the pointer to the next object.

⑤ if a virtual fu^n defined in a base class it need not necessarily redefined in the base class. in such cases calls will invoke base fu^n.

- Pure Virtual function:

It is a normal practice to declare a fu^n virtual inside the base class and redefined it in the derived classes. The fu^n inside base class is seldom used for performing any task. it only serves as a placeholder.

eg: we have not defined any object of class "Dim" and therefore fun Area() in the base class has been desired empty. such fu^n called do nothing fu^n.

A DNF may defined as follow:

virtual void area() = 0;

such fu^n are called pure virtual fu^n. a pure virtual fu^n is a fu^n declared in a base class that has no definition relative to the base class. In such cases the compiler requires each derive class to either define a fu^n or redeclare it as a pure virtual fu^n remember that class containing pure virtual fu^n cannot be used to declare any objects of its own such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create base pointer required for achieving run time poly morphism.

```
class Dim
{
 protected :
  int w, h ;
 public :
  void setDim()
  {
   cout << "width:";
   cin >> w;
   cout << "Height :";
   cin >> h;
  }
 void display ()
 {
  cout <<" In W: " << w << " In H: " << h;
 }
```

```
    virtual void area() = 0; // pure virtual fun
};

class Rect : public Dim
{
  public:
    void area()
    {
      cout << "\n Rect area :";
      cout << (w*h);
    }
};

class Trang : public Rect
{
  public:
    void area()
    {
      cout << "\n Triangle Area :";
      cout << (w*h)/2;
    }
};

int main()
{
  Dim *dptr;
  cout << "\n 1. Rect : \n 2. Triangle \n 3 : Exit \n
        choice :";
  int opt;
  cin >> opt;
  if (opt < 1 || opt > 2)
    return;
  if (opt == 1)
    dptr = new Rect;
  else
    dptr = new Trang;
  dptr -> set Dim();
  dptr -> display();
  if (opt == 1)
    cout << "\n Rect Area :";
  else
    cout << "\n Triangle Area :";
  dptr -> area();
  delete dptr;
}
```

```
    virtual void area() = 0; // pure virtual fun
};

class Rect : public Dim
{
  public:
    void area()
    {
      cout << "\n Rect area :";
      cout << (w*h);
    }
};

class Trang : public Rect
{
  public:
    void area()
    {
      cout << "\n Triangle Area :";
      cout << (w*h)/2;
    }
};

int main()
{
  Dim *dptr;
  cout << "\n 1. Rect : \n 2. Triangle \n 3 : Exit \n
        choice :";
  int opt;
  cin >> opt;
  if (opt < 1 || opt > 2)
    return;
  if (opt == 1)
    dptr = new rect;
  else
    dptr = new Trang;
  dptr -> set Dim();
  dptr -> display();
  if (opt == 1)
    cout << "\n Rect Area :";
  else
    cout << "\n Triangle Area :";
  dptr -> area();
  delete dptr;
}
```

FILE HANDLING IN C++ :

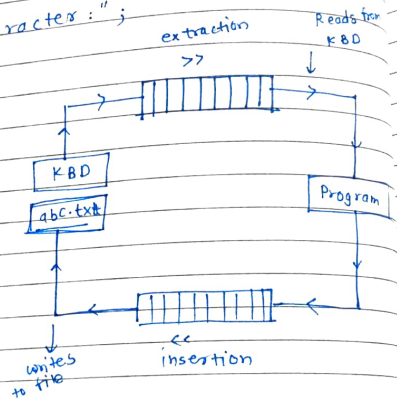
① ofstream => class object is used to write to file.

② ifstream => class object is used to read from file

③ fstream => class object is used to read / write from / to file.

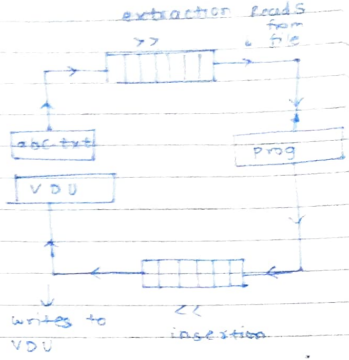
- Program to create text file abc.txt. :
  - enter * character to stop writing to the file.

```
#include<fstream.h>
void main()
{
ofstream out("abc.txt");
char ch;
cout<< "\n Type character :";
while(1)
{
cin>> ch;
if(ch=='*')
  break;
out<<ch;
}
out.close();
return;
}
```



- program to read a text file abc.txt

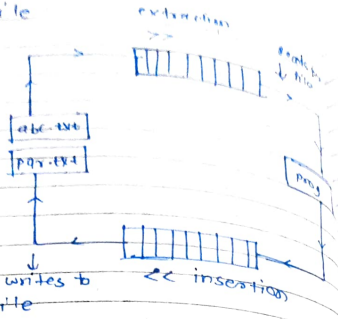
```
void main()
{
ifstream in("abc.txt");
char ch;
cout<<"\n File data :";
while(1)
{
  in>>ch;
  if(in.eof())
    break;
  cout<<ch;
}
in.close();
return;
}
```



- write a program to copy abc.txt file with new name pgr.txt.

```
void main()
{
ifstream in("abc.txt");
ofstream out("pgr.txt");
char ch;
cout<<"\n File Data :\n";
while(1)
{
  in>>ch; // read from file
  if(in.eof())
    break;
```

```
out << ch ; // write to file
}
in.close(); out.close();
return;
}
```



- program to create text file containing student records

```
void main()
{
 ofstream sout("stu.txt");
 int no;
 char nm[15];
 double mrk;
 while(1)
 {
  cout<<"\n Roll no: ";
  cin>>no;
  if(no==0)
   break;
 cout<<"\n Name";
 cin>>nm;
 cout>>"\n Marks: ";
```

```
 sout<<no<<" "<<nm<<" "<<mrk<<"\n";
 }
 sout.close();
 }
```

- write a program to read a file "stu.txt"

```
void main
{
 ifstream sin("stu.txt");
  int no;
 char nm[15];
 double mrk;
 while(1)
 {
  sin>>no>>nm>>mrk;
  if(sin.eof())
     break;
  cout<<no<<" "<<nm<<" "<<mrk<<endl;
 }
 sin.close();
 return;
 }
```

- Binary File Manipulation:

a) Function to write class object to file.

- <file_obj>.write((char *) &obj, sizeof(obj));

b) Function to read class object from file.

<file_obj>.read((char *) &obj, sizeof(obj));

(A) Create a header file and define class "stu.h"

```
class stu
{
 int no;
 char nm[15];
 double mrk;
public:
 stu ()
 {
 no=0; nm[0]='\0';
 mrk=0.0;
 }
 ~stu () { }
friend istream &operator >> (istream &in, stu &t).
{
cout << "\n Roll NO:";
in >>t.no;
cout << "\n Name : ";
in >> t.nm;
cout << "\n Marks: ";
in >> t.mrk;
return in;
}
friend ostream &operator <<(ostream &out, stu &t)
{
out<<"\n Roll NO: "<< t.no <<"\n Name : "<< t.nm <<
"\n Marks: "<< t.mrk;
return out;
}
int getNO()
{
return no;
}
double getMarks () { return mrk; }
};
```

(B) program to create a binary file and write "stu" class objects.

```
#include <stu.h>
void main ()
{
 ofstream sout;
sout.open ("stu.dat", ios::out | ios::binary
 stu obj;
 int no = 1;
 while (1)
```

writing

```
{
    cin >> obj;
    sout.write((char *) &obj, sizeof(obj));   // write class obj to file
    cout << "\n Enter 0 to Exit \n choice: ";
    cin >> no;
    if (no == 0)
        break;
}
sout.close();
return;
}
```

- program to read the binary file containing class objects.

```
#include "stu.h";
void main()
{
    ifstream sin;
    sin.open("stu.dat", ios::in | ios::binary);   // in: reading
    stu obj;
    clrscr();
    while(1)
    {
        sin.read((char *) &obj, sizeof(obj));   // read class object from file
        if (sin.eof())
            break;
        cout << obj;
    }
    sin.close();
}
```

- Opening files using open method:

If we want to use disk file we need to decide following things about the file and its d intended use.

i) suitable name for file.
ii) Data type and structure.
iii) purpose.
iv) Opening method.

The file name is string of characters that make up valid file name for operating system.

for opening a file we must first create a file stream and then link it to the file name. d file stream is defined using the classes ifstream, ofstream and fstream that are contained in the header file fstream.h. the classes to be used depends upon the purpose. i.e. whether we want to read from the disk or write data to it. a file can be opened in two ways. ① Using constructor fu^n of the class. ② Using member fu^n open of the class. The 1st method is useful when we use only one file in the stream. The 2nd method is used when we want to manage multiple files using one stream.

The connection with a file is closed automatically when the stream object expires otherwise we can make use of close method to disconnect the file from stream object.

- Opening files using open() method:

The fun open() can be used to open multiple files that use the same object. In such cases we may create a single stream object and use it to open each file intern.

```
file_stream_class obj;
obj.open ("filename1");
---
obj.close();
obj.open ("filename2");
---
obj.close();
```

The above example opens 2 files in sequence for writing the data. note that the first file is closed before opening the 2nd one. This is necessary because a stream can be connected to only one file at a time. In diff. applications we may required use more than one file simultaneously in such cases we need to create no. of streamx lass objects and call open() method for each.

- Detecting end of file:

Detection of the end of file condition is necessary for preventing any further attempt to read data from the file and if stream class object.

eg: Sin returns a value of zero, if any error occurs in the file operation including the end of file condition. The eof fun is member fun of ios class it returns non-zero value if the end of file condition encountered and a zero otherwise.

File opening modes:

Here before we have used ifstream and ofstream constructors and the fun open to create new files as well as to open existing files. in both these methods we use only one argument that was file name. however these fun can take two arguments, the second one for specifying the file mode the general form of the fun open() with two arguments.

```
<stream_object>.open ("file_name", mode);
```

The 2nd argument mode specifies the purpose for which the file is opened. The following are the file mode parameters.

parameter	meaning
1) ios::app	append to end of file
2) ios::ate	go to the end of file on opening.
3) ios::binary	binary file.
4) ios::in	open file for reading only
5) ips::nocreate	open fails if the file does not exists.
6) ios::noreplace	open fails if the file already exists.

7) ios :: out — open file for writing only

8) ios :: trunc — delete contains of file if it exists.

- File pointers and their manipulation:

Each file has 2 associated pointers known as file pointers. one of them is called the input pointer or get pointer and the other is called the o/p pointer or put pointer. we can use these pointers to move to the files while reading or writing. The input pointer is used for reading the contents of given file location and the o/p pointer is used for writing to a given file location. Each time an input or out put operation takes place the appropriate pointer automatically advance.

Default Actions:

When we open a file in read only mode the input pointer is automatically set at the beginning when we open a file in write only mode the output pointer is set at the begining. Incase we want to open an existing file to add more data the file is opened in append mode this moves out put pointer to the end of the file.

The above actions on file pointer takes place automatically. If we want to move a file pointer to any other desired location then we need to use following fun^

(1) seekg() → moves get pointer to a specified location.

(2) seekp() → moves put pointer to a specified location

(3) tellg() → gives current position to a get pointer.

(4) tellp() → gives current posin to a output pointer.

Both seekg() and seekp() fu: uses two argument.

1) seekg (<byte_no>, <ref-position>);
   seekp (<byte_no>, <ref-position>);

The parameter <byte_no> represent the no. of bytes. The file pointer is to be moved from the location specified by the reference position. The ref-position takes one of the following three constants defined in the ios class:

ios :: beg ⇒ starts from the begin.

ios :: cur ⇒ current posin of the pointer.

ios :: end ⇒ end of the file.

- Program to create binary file "acc.dat" containing account class objects with following cond^n.
  i) account no. must be unique.
  ii) account balance must be above 500.

```
class Account
{
  int no;
  char nm[15];
  int opnbal;
  int state;   state = 0 ⇒ closed acc. , state = 1 ⇒ valid acc.
public:
  Account()
  {
    no = opnbal = 0;
    state = 1;
    nm[0] = '\0';
  }
  ~Account(){}
  void setData(int n);
  void display();
  int getNO(){ return no; }
  int getState(){ return state; }
  int getBal(){ return opnbal; }
  void delRecord()
  {
    state = 0;
  }
};
```

```
void Account :: setData(int n)
{
  no = n;
  cout << "\n Name:";
  cin >> nm;
  cout << "\n Opening Balance:";
  do
  {
    cin >> opnbal;
  } while(opnbal < 500);
  state = 1;
}
void Account :: display()
{
  cout << "\nNO:" << no << "\n Name:" << nm <<
  "\n Opening Bal:" << opnbal << "\n State:" << state;
}

class AcctAMD
{
  Account obj;
  fstream fa;
public:
  AcctAMD();
  ~AcctAMD()
  {
    fa.close();
  }
  int search(int); // return -1 if record not found otherwise Rno
  void add();
```

```
    void mod();
    void del();
    void display();
    Account getobject()
    {
      return obj;
    }
};

AcctAMD::AcctAMD()
{
  ifstream in;
  in.open("acct.dat", ios::in | ios::binary);
  if(!in) // file open failed
  {
    ofstream out;
    out.open("acct.dat", ios::out | ios::binary);
    out.close();
  }
  else
    in.close();
  fa.open("acct.dat", pos::in | ios::out | ios::bin
}

int AcctAMD::search(int n)
{
  int pos=0;
  fa.seekg(0, ios::beg); // shifts get ptr to 0th
                         byte from begin

while(1)
{
  fa.read((char*)&obj, sizeof(Account));
  // reads an object from file
  if(fa.eof())
  {
    fa.clear(), // at the end of reading the current contents
    pos=-1;     of the file the program sets EOF flag on this
    break;      ~~moments any further reading from or writing~~
  }             ~~to the file the EOF flag is turned off by using~~
  if(obj.getNo()==n)  the fn clear() which allows access to the file
    break;            once again.
  pos = fa.tellg(); // tells present get pointer pos in
}                                  in bytes from begin.
return pos;
}
```