Object Oriented Programming in C++

CHAPTER 01 – Introduction to OOP & C++

Difference between Procedure Oriented and Object Oriented Programming

Procedure Oriented Programming	***Object Oriented Programming***
1. Emphasis is on Functions	Emphasis is on Data
2. Problem is solved using Functions	Problem is solved using Objects
3. Data is freely available, that can be accessed by any function	Data is encapsulated in side the Object
4. Any Function can manipulate a common Data	Only member functions inside the Object can manipulate the Data of the Object
5. There is no binding between Data and Functions	There is strong binding between Data and Functions inside an Object
6. Only local data inside Function is protected from access from outside the Function	Data encapsulated inside Private access specifier in a Class, can not be accessed from outside the Class.
7. It follows Top-Down approach towards problem solving (emphasis on Functions then Data)	It follows Bottom-Up approach towards problem solving (emphasis on Data then Functions)

Basic Object Oriented Concepts

1. ***Object***
2. ***Class***
3. ***Encapsulation***
4. ***Abstraction***
5. ***Data Hiding***
6. ***Inheritance***
7. ***Polymorphism***
8. ***Dynamic binding***
9. ***Message Passing***

1. ***Object :-***
    1. An Object is an Instance of a Class. An Instance is the existence in the computer system by acquiring a memory space.
    2. An Object is the only way a Class becomes usable.
    3. Objects are basis of Object Oriented Programming.
    4. An Object has all the Characteristics of the Class using which the Object is created.
    5. It implements reusability of code written in Classes.
2. ***Class :-***
    1. A Class is a Blueprint or Stencil for creating Objects.
    2. A Class specifies all the Member Data and Member Functions that would be present in the Objects created using the Class.
    3. Using a Class any number of Objects can be created.
    4. It's a User Defined Data Type also called Abstract Data Type, which acts like a basic data type when used.
    5. Usually Classes are used to represent computer understandable formats for Real World Entities.
3. ***Encapsulation :-***
    1. It's the process of wrapping up of Data and Functions inside a single Entity (usually Classes).
    2. Encapsulation helps to establish Abstraction and Data Hiding.
    3. Functions, Structures and Classes implement Encapsulation at programmer level.
4. ***Abstraction :-***
    1. It's the process of Hiding the complexities of implementation and providing a simple Interface for easy use.
    2. Its implemented using Encapsulation.
    3. Functions, Structures and Classes implement Abstraction at programmer level.
5. ***Data Hiding :-***
    1. It's the process of keeping the Data under such an Access mode that its only accessible to permitted Functions.
    2. Using Private, Protected specifiers, we can hide Data from access from outside a Class.
    3. Functions, Structures and Classes implement Data Hiding at programmer level.
6. ***Inheritance :-***
    1. It's the process of Passing Attributes (data members) and Characteristics (member functions) of one Class to other Classes.
    2. Here the Class which gets the Inherited properties is called the Child Class and from which it acquires the properties is called the Base or Parent Class.
    3. During Inheritance the Protected and Public Data and Functions get passed on as copies from the Parent or Base Classes to Child Classes.
    4. It forms a Hierarchical Structure from Parent down to Children Classes.

5. It implements Reusability of Parent Class Data and Functions.
6. Structures and Classes can be used to implement Inheritance.
7. Types :- Single, Multilevel, Multiple, Hierarchical and Hybrid.

7. Polymorphism :-

1. It's the process of defining more than one kind implementation (coding) of a function, using same name with either Different number of Arguments or Different Data types of Arguments.
2. Implementation :- Function Overloading, Constructor Overloading and Operator Overloading and Templates.
3. Types:- Compile Time (the compiler recognizes the binding between the function and its code during compile time) and Run Time(the compiler recognizes the binding between the function and its code during runtime)

8. Dynamic binding :-

1. Its also called Runtime Polymorphism, where the compiler recognizes the binding between the function call and its code during runtime.
2. Its implemented using Virtual Functions, during Inheritance.

9. Message Passing :-

1. It's the process of passing Data between Objects of same Class and also of different Classes.
2. Its implemented using Function calling or by simply sharing data.
3. Types:- From User to Objects (done by invoking Function inside an Object) and From Object to Object (by sharing Data of one Object with another Object).
4. Using Friend functions (with objects as input arguments), message passing can be done between two or more Objects of Different Classes.
5. Using member Functions (with objects as input arguments), message passing can be done between objects of same Class.

<u>Difference Between Pointer and Reference Variables</u>

Pointer Variable	***Reference Variable***
1. It is a special variable which can store the address of another variable	It is a duplicate name (alias) given to any variable.
2. Here the Pointer variable will be created in the memory and then it can be assigned any address.	No extra space is required. Only a duplicate name is assigned to the same memory area.
3. Using **** or de-referencing*** operator along with the pointer variable the value inside the address can be accessed	By simply using an ***= or assignment operator*** with the reference variable the value can be changed
4. Used while implementing Call By Address	Used while implementing Call By Reference
5. Common Syntax :- ***Declaration :-*** <u>Data type *Pointer variable name ;</u> ***Acquiring address :-*** <u>Pointer variable = & variable name ;</u> Here the address of the ‘ variable ‘ would be stored get stored in the ‘ pointer variable ‘. ***Manipulating data :-*** *Pointer variable = value; here the ‘ value ‘ would be assigned to the ‘ variable name ‘.	***Common Syntax :-*** ***Declaration :-*** <u>Data type & reference = variable ;</u> Here the LHS ‘ reference ‘ name given would become the duplicate name for the ‘ variable ‘ in the RHS ***Manipulating data :-*** <u>reference = value ;</u> here the ‘ value ‘ would be assigned to the ‘ variable ‘, which would be reflected in both the reference as well as the original ‘ variable’.

6. Example:-
int a = 10 ;
int *p ;
cout<< a ; // it would show 10
p = &a; // p is pointing to a
*p = 20;
cout<< a ; // it would show 20
	Example :-
	int a = 10;
	int &b = a; // b is duplicate name of a
	cout << a << b ; // It would show 10 10
	b=20;
	cout << a << b ; // It would show 20 20

Pointer Variable	***Reference Variable***

7. Pointer to an Array
data-type array[SIZE];
data-type * ptr=array;
 	
ex:
int ar[5];
int * p=ar;
for(int I=0;I<5;I++)
cin>>p[I];
	Reference to an Array
	data-type array[SIZE];
	data-type * &ref=array;
	
	ex:
	int ar[5];
	int * &r=ar;
	for(int I=0;I<5;I++)
	cin>>r[I];

8. Pointer to Pointer
ex:
int a=10;
int *p1=&a;
int **p2=&p1;
cout<<a<<*p<<**p; // 10 10 10
**p=5;
cout<<a<<*p<<**p; // 5 5 5
a=3;
cout<<a<<*p<<**p; // 3 3 3
	Reference to Reference
	ex:
	int a=10;
	int &r1=a;
	int &r2=r1;
	cout<<a<<r1<<r2; // 10 10 10
	r2=5;
	cout<<a<<r1<<r2; // 5 5 5
	a=3;
	cout<<a<<r1<<r2; // 3 3 3

Benefits of OOP

1. Implementing Reusability by making Reusable Classes and creating Objects from those Classes.
2. Providing various levels of Data protection and hiding by using Public, Private and Protected access specifiers.
3. Providing Protection to Data and Functions both.
4. Helps in creating Flexible, Extensible and Maintainable code.
5. Effective solutions for Real World problems by implementing Classes representing Real World Entities.
6. Quick software development due to Reusability and easy implementation of Real World Entities and their Relationships using Inheritance.

The Scope Resolution Operator ::

1. The Scope Resolution Operator is used to identify the value of variables referred, which are declared in different scopes.
2. Its used to assign initial values to a Static variable of a Class.
3. Its used for defining the Member Functions outside the Class.
4. The Scope Resolution Operator is represented using ***::*** symbol.
5. ***Syntax*** :-
   a. Scope Name ***::*** variable name ;
   b. Data type Scope Name ***::*** static variable name ;
   c. Return type Scope Name ***::*** function name (input argument list)
      { definition of the function }

6. ***Example***:-

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

class A
{
  static int s;
  public:
  int a;
  void function();
};
int A :: s=1;
void A :: function()
{
 cout<<"Welcome to my Function definition";
}

int a=10;

main()
{
 clrscr();
 int a=200;
 cout<<"\n"<<a; // output  200 (local a)
 cout<<"\n"<<::a; // output  10 (global a)
 // cout<<A::a; // cannot be done with out creating an object;
 getch();
}
```

<u>Difference Between New and Delete Operators</u>

New Operator	***Delete Operator***
1. Its used to allocate memory space for the desired data-type.	Its used to free (De-allocate) the memory spaces allocated by New operator.
2. It returns the Address of the newly reserved (allocated) Memory space.	It does not.
3. It accepts as an argument the Data Type for which the memory has to be Reserved.	It accepts the Pointer variable as an argument.
4. There are Three different ways of using New operator :- a) To allocate single memory space with no default value set. b) To allocate single memory space with a default value set. c) To allocate an array of memory spaces.	There are Two ways of using Delete operator :- a) De-allocating the pointer, which is pointing towards the reserved (allocated) memory space or spaces. b) De-allocating all the memory spaces, which is reserved by the Pointer.

5. Common Syntax :-
Single allocation :-
Pointer-variable = ***new*** data-type ;
Pointer-variable = ***new*** data-type(value) ;
Array allocation :-
Pointer-variable = ***new*** data-type [size] ;
	Common Syntax :-
	Single De-allocation :-
	delete Pointer-variable ;
	Array allocation :-
	delete [ ] Pointer-variable ;

6. Example :- int *p = new int ; int *q = new int (10) ; int *ar = new int [20] ;	***Example :-*** delete p ; delete [ ] p ;

Functions in C++

1. ***Functions are used to implement reusability of code.***
2. ***There are Three steps of writing Functions :-***
   1. ***Function Declaration :-*** Here we write the Function prototype, which represents the name of function, number of input arguments and data-types of input arguments.

      Syntax :- return-data-type function-name (input-argument-list along with their data-types) ;

      Example :- int add (int, int) ;
   2. ***Function Definition :-*** Here we use along with the prototype, Opening and Closing curly brackets, inside which we write the code of the function.

      Syntax :- return-data-type function-name (input-argument-list along with their data-types) { valid C++ codes }

      Example :- int add (int a, int b) { return (a+b) ; }
   3. ***Function Calling :-*** Here we use the function prototype, with out the return type and passing in the real arguments. It causes the function to execute its code.

      Syntax :- function-name (argument-values) ;

      Example :- add (10 , 20) ;
3. ***There are Three categories of Calling Functions in C++***

1. ***Call by Value :-***
   1. Here we pass values in the input arguments space, during function calling.
   2. The receiving variables in the argument list of the function definition are Ordinary variables.
   3. Example :-

```
void  show(int a)
{
     cout << a ; // variable 'a' has a copy of the Value of 'b'.
}
main()
{
     int b=20 ;
     show(10) ; // output 10
     show(b) ; // output 20
}
```

2. ***Call by Address :*** -
   1. Here we pass Address as values in the input argument space, during function calling.
   2. The receiving variables in the argument list of the function definition are Pointer variables.

3. Example :-

```
void  change(int *p)
{
     *p = 100 ; // pointer 'p' is pointing towards the Address where variable
'a' is present.
}
main()
{
     int a=20 ;
     cout<< a ; // output 20
     show(&a) ; // passing address of 'a'
     cout<< a ; // output 100
}
```

3. Call by Reference :-

1. Here we pass the variable name in the input argument space, during function calling.
2. The receiving variable in the argument list of the function definition is Reference variable.
3. Example :-

```
void  change(int &r)
{
     r = 100 ; // 'r' is a reference to variable 'a'
}
main()
{
     int a=10 ;
     cout<< a ; // output 10
     show(a) ; // passing 'a'
     cout<< a ; // output 100
}
```

4. ***There are Three categories of Returning from Functions in C++***

1. ***Return by Value :-***
    1. Here the value returned by the function after it is executed is a simple value.
    2. Example :-

```
int  greater(int a, int b)
{
     if ( a > b)
     {
             return a ;
     }
     else
     {
             return b ;
     }
}
main()
{
     int i, j ;
     cin >> i >> j ;
     cout << " Greater Value is:- " << greater ( i , j );
}
```

2. ***Return by Address : -***
    1. Here the returned value is an address, and has to be received in a pointer.
    2. Example :-

```
int*  fill (int *p)
{
     p = new int [5] ;
     for ( int j = 0 ; j < 5 ; j ++ )
     {
             cin >> p [ j ] ;
     }
     return p ;
}
main()
{
     int *q;
     q = fill ( q ) ;
     for ( int k = 0 ; k < 5 ; k ++ )
     cout<< q [ k ] ;
}
```

3. Return by Reference :-

1. Here value returned is the reference to a variable.
2. Example :-

```
int&  greater(int & r, int & j)
{
    if ( r > j )
    {
            return r ;
    }
    else
    {
            return j ;
    }
}
main()
{
    int  a = 10, b = 5 ;
    greater ( a , b ) = 100 ; // Here the reference to the greatest of a and b
will be returned and will be assigned the value 100
    cout<< a << b ; // output  100  5
}
```

5. Inline Functions :-

1. By default every function created in C++ is Inline.
2. These functions are expanded in line, just like Macro expansions. The function definition code is placed in place of the calling of the function during compile time by the compiler. But the difference is that in Macro there is no Type-Checking, but in case of Functions there is implicit Type-Checking.
3. To make any Function Inline we can explicitly mention the keyword " inline " before function declaration.
4. Syntax :- <u>inline return-type function-name (input argument list) { coding for function definition }</u>
5. Example :-

```
inline   void   myfun (void)
{
    cout<<"Welcome to my inline function" ;
}
main()
{
    myfun( ) ;
}
```

6. Situations where any Inline Function does not remain Inline although declared Inline.
    1) When a return statement is used.
    2) When loops, switch or goto statements are used.
    3) When static variables are used.
    4) When recursion is used.

6. Default Arguments :-

1) Its the default value given to the parameters in a function.
2) The default values get used only during calling of the function when we don't provide the values to those parameters.
3) The default values can only be given from right most onwards towards left.
4) The default values can not be started from left or middle.
5) The default values are given in the function definition.
6) Syntax :-

return-type function-name (argument1 = value1 , argument2 = value2, …. , argument n = value n)
{ function definition code }

7) Example :-

```
void  add ( int a = 0 , int b = 0 )
{
    cout << ( a + b ) ; // output 5
}
main( )
{
    add( 5 ) ;        // only one argument is passed so the default value for the
                      // second argument  will be used
}
```

7. Const arguments :-

1) Here we declare the arguments as constants by using " const " keyword, in the function definition.
2) The arguments declared as const cannot be changed in side the function definition.
3) During calling of the function the const arguments would acquire the values initially given in the call.
4) Syntax :- return-type function-name (const argument, ….) { function definition code }
5) Example :-

```
void  add ( int a , const int b )
{
     a=10;
//   b=20 ;  // Not allowed as its declared const
     cout << ( a + b ) ; // output 15
}
main( )
{
     add( 5 , 5 ) ;
}
```

8. Function Overloading :-

1) It's a form of Polymorphism.
2) Here we can create more than one definition of function with same Name. But the number of input arguments should be different for all the definitions or data-type of input arguments should be different.
3) According to the type or number of input arguments given during calling of the function, the compiler binds the specific definition with specific call of function.
4) Example :-

```
void  add ( int a , int b )
{    cout << ( a + b ) ; // output 10
}
void  add ( int a , int b, int c )
{    cout << ( a + b + c ) ; // output 15
}
main( )
{
     add( 5 , 5 ); // the two argument add function will be called
     add( 5 , 5 , 5 ); // the three argument add function will be called
}
```

CHAPTER 02 – Class , Nested Class, Const, Static members and Friend Functions

Class :-

1. It's a User Defined Data-type.
2. It implements Data-Hiding, Encapsulation and Abstraction.
3. The Data declared in a Class are called Data-Members of the Class.
4. The Functions declared or defined in a Class are called Member-Functions of the Class.
5. Basic syntax for Class :-

```
Class class-name
{
        declarations of data members
        and functions.
        Functions can be defined here itself, but not the data members.
};
```

6. The Member Functions can be defined inside the Class. Using the Scope-Resolution-Operator ‘ ***::*** ‘ they can be defined outside the class.
   - Syntax :-

   Return-type Class-Name ***::*** Function-Name (Input-Argument-List) { Definition codes for the function }
7. The members of a Class can only be accessed through Objects of the class.
   - ***For simple Objects :-*** The members can be accessed by using ‘ ***.*** ‘ operator along with the object.
     - Syntax :- Object-Name ***.*** member-name ;
   - ***For Pointer Objects :-*** The members can be accessed by using ‘ ***->*** ‘ operator along with the object.
     - Syntax :- Pointer-Object-Name ***->*** member-name ;
8. Access Specifiers are used to represent the visibility or accessibility of any member outside the Class.
9. There are Three Access Specifiers :-
   - ***Public :-***
     - Any member declared under this specifier is Accessible from Outside the class, by using the object of the Class.
     - The Public members of a Class get Inherited completely to the Child Classes.
     - The keyword “ ***public :*** “ is used.

- Syntax :-

  ```
  Class definition
  {
          public :
          declarations or definitions
  } ;
  ```

- Example :-

  ```
  class A
  {
          public :
          int a;
  } ;
  main()
  {
          A  oba ;
          oba . a = 10 ; // we have set the value of variable  ‘ a ‘
  in object  ‘ oba ‘ to 10
  }
  ```

- ***Private :-***
  - Any member declared under this specifier is Not Accessible from Outside the Class.
  - The Private members are accessible to only the Member Functions and Friend Functions in the Class.
  - The Private members of a Class Never get Inherited to the Child Classes.
  - The keyword “ ***private :*** “ is used.
  - Syntax :-

    ```
    Class definition
    {
            private :
            declarations or definitions
    } ;
    ```

- Example :-

  ```
  class A
  {
          private :
          int a;
          public :
          void fill ( )
          {
                  cin >> a ;
          }
  } ;
  main()
  {
          A  oba ;
  //      oba . a = 10 ; //  invalid as  ‘ a ‘ is private member.
          oba . fill ( ) ; //  valid as  fill ( )  is a public member.
  }
  ```

- ***Protected :-***
  - Any member declared under this specifier is Not Accessible from Outside the class, by using the object of the Class.
  - The Protected members of a Class get Inherited completely to the Child Classes and they maintain their Protected visibility in the child Class.
  - The Protected members are accessible to only the Member Functions and Friend Functions in the Class.
  - The keyword “ ***protected :*** “ is used.
  - Syntax :-

    ```
    Class definition
    {
            protected :
            declarations or definitions
    } ;
    ```

- Example :-

```
class A
{
          protected :
          int a;
          public :
          void fill ( )
          {
                    cin >> a ;
          }
} ;
main()
{
          A  oba ;
//        oba . a = 10 ; //  invalid as  ‘ a ‘ is protected member.
          oba . fill ( ) ; //  valid as  fill ( )  is a public member.
}
```

10. By default (if no access specifier is mentioned then) the members of a class become private members.
11. Using Class we can create our own data type and objects created from our class will contain a copy of all the members present in the class definition.

Nested Class :-

1. Nested Classes are the Classes defined inside a Class.
2. Neither the Outer Class nor the Inner Class can access each others data.
3. Objects created from Outer Class can have access to only the member data and member functions of the outer class.
4. Objects created from Inner Class can have access to only the member data and functions of the inner class.
5. Access specifiers does not effect the visibility of Nested Classes.

6. Example:-

```
class A//outer class
{
  public:
  class B          // definition of nested class - inner class
  {
   int a;
   public:

   void fill();
   void show()
   {
    cout<<"\n"<<a;
   }
  };
};
void A :: B :: fill()
{
cin>>a;
}

main()
{
  clrscr();
  A :: B b ; // Nested class B object created
  b..fill();
  b.show();
  getch();
}
```

Static Members :-

1. Only a single copy of the Static members are shared between Objects of the Class.
2. We can create Static Data Members and Static Member Functions.
3. Static Member Functions can have access to Only Static Members in that Class.
4. Static Members are called using the Class-Name and Scope-Resolution-Operator and Member-Name
5. The Static members are declared using " ***static*** " keyword.
6. Syntax :-
   - For functions :- static return-type function-name (input arguments) { function definition codes }
   - For variables :- static data-type variable-name ;
7. Example:-

```
class A
{
 static int a;
 public:
 static void myfun()
 {
  a=10;
  cout<<"\n"<<a;
 }
 A()
 {
  a+=1;          // manipulating the static member data which
                 //would remain updated for all the objects created for this
                 //class.
  cout<<"\nObjects created :- "<<a;
 }
};
int A::a;        // default initialization to Zero value
                 // The initialization process is Mandatory
main()
{
 clrscr();
 A oba1;                // output:- Objects created :- 1
 A oba2;                // output:- Objects created :- 2
 A::myfun();            // calling of the static member function
 getch();
}
```

Friend Functions :-

1. Friend functions are Not member functions of any class, but they can access the private, protected and public members of the class.
2. They are declared using " ***friend*** " keyword.
3. They Don't belong to any specific Class.
4. They can be invoked as ordinary functions, with out using any object.
5. They can be defined outside the Class, here also they are defined like ordinary functions and Not like member functions.
6. The access specifiers does not effect the visibility of Friend Functions.
7. Syntax :-
```
friend   return-type  function-name ( input-arguments ) ;
```
8. Example :-
```
class B ;  // Forward declaration to satisfy the compiler about class B
class A
{
int a ;
friend  void  swap ( A ob1, B ob2 ) ;
public:
void fill(int n)
 {   a = n ; }
void show()
{ cout<<a; }
};


class B
{
int b ;
friend  void  swap ( A ob1, B ob2 ) ;
public:
void fill(int n)
{   b = n ; }
void show()
{ cout<<b ;}
};
```

```
void swap ( A &ob1, B &ob2 )
{
  int temp;
  temp = ob1 . a ;
  ob1 . a = ob2 . b ;
  ob2 . b = temp ;
}
main()
{
 clrscr();
 A  ob1;
 B  ob2;
ob1.fill(10);
ob2.fill(20);
 swap( ob1 , ob2) ;
ob1.show();
ob2.show();
 getch();
}
```

Friend Class :-

1. A friend class is a class declared as friend in another class.
2. " friend " keyword is being used to declare a class as friend.
3. Access specifiers do not effect the declaration.
4. When a class C1 is declared as friend in class C2. Then from class C1 all the members of class C2 can be accessed with the help of an object of class C2.

5. Example:-

```
class C1;
class C2
{
 int a;                          // All the declarations/definitions are in Private mode
void fill() {cin >> a;}
void show(){ cout << a;}
friend B;                                // or can be declared as:- friend class B;
};
class C1
{
 C2 obc2;
 public:
void fillshow()
{
 obc2.fill();
 obc2.show();
}
};
main()
{
 C1 obc1;
 obc1.fillshow();
}
```

CHAPTER 03 – Constructor, Destructor and Operator Overloading

Constructors :-

1. It's a special function which has the same name as that of the Class name.
2. It has No Return-Type, not even void.
3. Constructor overloading is possible.
4. Their accessibility gets effected by the Access specifiers.
5. They can be Friend.
6. Default arguments are possible.
7. They are invoked automatically as soon as Objects of Class are created.
8. We can explicitly call Constructors of a Class.
9. They can Not be Inherited.
10. There are Three Basic Categories of Constructors
    a. Default Constructor
       Syntax :- class-name ( ) { code }
    b. Parameterized Constructor
       Syntax :- Syntax :- class-name (parameter-list) { code }
    c. Copy Constructor.
       Syntax :- class-name (reference of the same class) { code }
11. For every Class defined the compiler creates a Default Constructor and Copy Constructor by default, implicitly.
12. If in any Class the Default Constructor is not defined then the compiler calls the implicit Default Constructor for Initializing objects of the Class. If there exists a Default Constructor defined in the Class then the defined Constructor will be called.
13. If any of the three categories of Constructors are defined in the Class then the compiler in any case would never use the implicitly created Constructor, it would only use the user defined Constructor.
14. When an Object Initialization (assignment during declaration) is done using another Object (also called Copy Initialization) then the Copy Constructor is invoked.
15. When an Object is passed using Call-By-Value then also the Copy Constructor is invoked.
16. Constructors of a class can be invoked by using existing objects of the class.
    Syntax :- obect . class-name :: constructor-function-call ;
17. Example :-

```
class A
{
 int a;
 public:
 A()              //Default Constructor
 {
  cout<<"Default Constructor Called";
 }
 A(int z)         //Parameterized Constructor
 {
  a=z;
  cout<<"Parameterized Constructor Called";
 }
 A(A &r)          //Copy Constructor
 {
  a=r.a;
  cout<<"Copy Constructor Called";
 }
 void show()
 {
  cout<<"\n"<<a;
 }
};
void byVal(A ob)        //Call by Value
{
}
void byRef(A &r)        //Call by Reference
{
}
```

```
main()
{
 clrscr();
 A ob1;          //Implicit call    //Calls the Default Constructor
 A ob4 = A( ) ; //Explicit call    //Calls the Default Constructor
 A ob2(10);     //Calls the Parameterized Constructor
 A ob3=ob2;     //Calls the Copy Constructor (Copy Initialization)
 ob2=ob1;       //Does Not Call our defined Copy Constructor
 ob2. A :: A (20) ;       //Explicitly calling the Parameterized Constructor
 byVal(ob2);    //Calls the Copy Constructor
 byRef(ob2);    //Does Not Call the Copy Constructor
 getch();
}
```

Destructors :-

1. It's a special function which has the same name as that of the Class name.
2. It has No Input-Arguments and No Return-Type, not even void.
3. Its definition is preceded by Tild symbol " ~ ".
4. Its called implicitly as soon as the scope for any object finishes. Its invoked for every object .
5. The compiler creates a Destructor for every Class implicitly by default.
6. If a Destructor is explicitly defined then the compiler will always use the explicitly defined Destructor.
7. Its generally used to free up the spaces allocated by using " new " operator during calling of Constructors. The de-allocation should be done using " delete " operator
8. The object that is created First is Destructed Last and object created Last is Destructed First.
9. Syntax :- ~class-name ( ) { code }

10. Example :-

```
class A
{
 static int cnt;
 int id;
 public:
 A()               //Constructor
 {
  cnt+=1;
  id=cnt;
  cout<<"\n"<<" Object Created Id = "<<id;
 }
 ~A()              //Destructor
 {
  cout<<"\nDestructor Called for Object Id = "<<id;
 }
};
int A::cnt;

main()
{
 clrscr();
 {
  A ob1;           //Created First Destructed Last
  A ob2;           //Created Second Destructed Last Second
  A ob3;           //Created Last Destructed First
 }                 //As scope is finishing so Destructors would be called
 getch();
}
```

Operator Overloading :-

1. It's a type of Polymorphism, where we can give additional meaning or implementation to existing operators.
2. But the usage of Overloaded Operators should follow the usual rules laid down by the compiler.
3. We define the overloaded operator as a function with "operator" keyword attached before the operator-symbol.
4. When using Type-Casting operators the following points need to be remembered :-
    a. It can Not be declared as Friend, so it has to be a member function.
    b. No return-type not even void.
    c. No input-arguments not even void.
    d. Should have a return statement for returning the required value.
    e. Type casting can be done as follows
        i. <u>Class type to Class type</u> :- Using overloaded casting operator of Class type.
        ii. <u>Class type to Basic type</u> :- Using overloaded casting operator of Basic type.
        iii. <u>Basic type to Class type</u> :- Using Single Parameter Constructor
    f. Example:-

```
class A
{
 float a;
 friend B;         //Otherwise temp cannot access a
 public:
 void fill()
 {
  cin>>a;
 }
 void show()
 {
  cout<<"\n"<<a;
 }
};
```

```
class B
{
 int b;
 public:
 B()
 {
 }
 operator A()              //Type Convertion from Class type to Class type
 {
  A temp;
  temp.a=b*0.1;
                      //temp can access a as classB is friend of classA
  return temp;
 }
 operator char()           //Type Convertion from Class type to basic Type
 {
  return ((char)b);
 }
 B(float z)                //Type convertion from basic Type to Class type
 {                        //Here we need to use a single parameter Constructor
   b=z;
 }
 void fill()
 {
  cin>>b;
 }
 void show()
 {
  cout<<"\n"<<b;
 }
};
```

```
main()
{
 clrscr();
 A ob1;
 B ob2;
 ob2.fill();
//Type convertion from classB type to classA type
 ob1=A(ob2);
ob1.show();
//Type convertion from classB object to char type
 cout<<"\n"<<char(ob2);
//Type convertion from float type to classB type
 ob2.B::B(22.5);
 ob2.show();
 getch();
}
```

5. There are two basic categories of operators that we can Overload
    a. Unary Operator
        - Syntax as member function :- return-type operator < symbol > ( ) { code }
        - Syntax as Friend function :-
          friend return-type operator < symbol > (one input argument) { code }
    b. Binary Operator
        - Syntax as member function :-
          return-type operator < symbol > (one input argument) { code }
        - Syntax as Friend function :-
          friend return-type operator < symbol > (two input arguments) { code }
6. Example :-

```
class A
{
 int a;
 public:
 A()
 { }
 A(int z)
 {
  a=z;
 }
 void operator - ()
 {
  a=-a;
 }
 void operator += (A ob)
 {
  a=a+ob.a;
 }
 void friend operator ++ (A &r)
 {
  r.a+=1;
 }
 A friend operator +(A ob1, A ob2)
 {
  A temp;
  temp = ob1.a + ob2.a;
  return(temp);
 }
void show()
 {
  cout<<"\n"<<a;
 }
};

main()
{
  clrscr();
  A ob1(10);
  A ob2(20);
  A ob3;
  -ob1;                        //equivalent calling ob1.operator-();
  ob1.show();                          // -10
  ob1+=ob2;                    //equivalent calling ob1.operator+=(ob2);
  ob1.show();                          // 10
  ob1++;                       //equivalent calling operator++(ob1);
  ob1.show();                          // 11
  ob3 = ob1 + ob2;             //equivalent calling operator+(ob1,ob2);
  ob3.show();                          // 31
getch();
}
```

7. List of operators that can Not be Overloaded :-
    a. Member Access Operators :- **.** & **.***
    b. Scope Resolution Operator :- **::**
    c. Size of Operator :- ***sizeof (data or data-type)***
    d. Conditional Operator (ternary operator) :- **? :**

Constant Members ;-

1. Constant Data members are declared using " const " keyword before the variable declaration.
2. Syntax :- const data-type constant-name ;
3. Constant member Functions are declared / defined using the " const " keyword after the closing parenthesis ') ' of function parameter list.
4. Syntax :- return-type function-name (parameter-list) const { function-definition-code }
5. Constant member Functions can Not modify the members of the Class.
6. Constant Data members can only be initialized using the Constructor of the Class.
7. Example :-

```
class A
{
 const int a;     // Constant member Data
 int b;
 public:
 A(int z):a(z)    // Initialization of Constant Data member
 {
 }
 int fill() const  //Constant member Function
 {
//   b=10;         // Cannot modify the data members
 }
 void show()
 {
  cout<<"\n"<<a ;
 }
};
main()
{
 clrscr();
 A ob(10);
 ob.show();
 getch();
}
```

CHAPTER 03 – Inheritance and Dynamic Binding

Inheritance :-

1. It's a method of implementing reusability of Classes.
2. The Members of one Class can be accumulated in another Class through Inheritance.
3. The Class which acts as the source of providing its Members for Inheritance is called the Parent or Base Class. The Class that derives from or acquires the Members of Base Class is called the Child or Derived Class.
4. The Private members of any Class can Not be Inherited.
5. The process of Inheritance can be categorized in to Three main Categories :-
    1) Public Inheritance :- Here the Child Class acquires the Protected and Public Members. And they Remain Protected and Public in the Child Class. Its done by using "public " keyword during Child Class definition.
       Syntax :-
       class child-class-name : public parent-class-name { definition of the child class} ;
    2) Private Inheritance :- Here the Child Class acquires the Protected and Public Members. And they Become Private in the Child Class. Its done by using "private " keyword during Child Class definition.
       Syntax :-
       class child-class-name : private parent-class-name
       { definition of the child class} ;
    3) Protected Inheritance :- Here the Child Class acquires the Protected and Public Members. And they Become Protected in the Child Class. Its done by using "protected " keyword during Child Class definition.
       Syntax :-
       class child-class-name : protected parent-class-name
       { definition of the child class} ;
6. There are Five Types of Inheritance Structures :-
    1) Single Level Inheritance
    2) Multilevel Inheritance
    3) Multiple Inheritance
    4) Hierarchical Inheritance
    5) Hybrid Inheritance
7. Single Level Inheritance :-
    1) Here there is one Base Class and one Derived Class.
    2) Syntax :- class child-class-name : <access-mode> parent-class-name { definition code for the child class } ;

3) Example :-

```
class A
{
 public:
 void display()
 {
  cout<<"\nClass A display() called";
 }
};
class B:public A
{
 public:
 void display()
 {
  cout<<"\nClass B display() called";
 }
};
main()
{
 clrscr();
 B ob;
 ob.display();
 ob.A::display();
 getch();
}
```

8. Multilevel Inheritance :-
    1) Here the Single Level Inheritance is extended to more than one level.
    2) Syntax :-

    class child1-class-name : <specifier> parent-class-name { child1 class definition } ;

    class child2-class-name : <specifier> child1-class-name { child2 class definition } ;

    3) Example :-

```
class A
{
 public:
 void display()
 {
  cout<<"\nClass A display() called";
 }
};
```

```
class B:public A
{
 public:
 void display()
 {
  cout<<"\nClass B display() called";
 }
};
class C:public B
{
 public:
 void display()
 {
  cout<<"\nClass C display() called";
 }
};
main()
{
 clrscr();
 C ob;
 ob.display();
 ob.B::display();
 ob.A::display();
 getch();
}
```

9. Multiple Inheritance :-
    1) Here the Child class is Inheriting more than one Parent classes at a time.
    2) Syntax :- class child-class-name : <specifier> parent1-class-name , <specifier> parent2-class-name { child class definition } ;
    3) Example :-

```
class A
{
 public:
 void display()
 {
  cout<<"\nClass A display() called";
 }
};
```

```
class B
{
 public:
 void display()
 {
  cout<<"\nClass B display() called";
 }
};
class C:public A, public B
{
 public:
 void display()
 {
  cout<<"\nClass C display() called";
 }
};
main()
{
 clrscr();
 C ob;
 ob.display();
 ob.B::display();
 ob.A::display();
 getch();
}
```

10. Hierarchical Inheritance :-

1) Here two or more Child classes Inherit the same Parent class.
2) Syntax :-

class child1-class-name : <access> parent1-class-name { child1 class definition } ;
class child2-class-name : <access> parent1-class-name { child2 class definition } ;

3) Example :-

```
class A
{
 public:
 void display()
 {
  cout<<"\nClass A display() called";
 }
};
```

```
class B:public A
{
 public:
 void display()
 {
  cout<<"\nClass B display() called";
 }
};
class C:public A
{
 public:
 void display()
 {
  cout<<"\nClass C display() called";
 }
};
main()
{
 clrscr();
 C ob1;
 B ob2;
 ob1.display();
 ob2.display();
 ob1.A::display();
 ob2.A::display();
 getch();
}
```

11. Hybrid Inheritance :-

1) It's a combination of single-level, multi-level, multiple and hierarchical Inheritance.
2) Here the resultant (last or grand-child) Child class in the structure can acquire duplicate copies from the grand parent class.
3) In situations where the resultant child class can acquire duplicate copies of the grand parent class then grand parent class used during inheritance is accompanied with " virtual " keyword, to avoid duplicacy in the grand child class.
4) Syntax :-

class parent1-class-name : virtual <access> grand-parent1-class-name
{ definition code for parent1-class } ;
class parent2-class-name : virtual <access> grand-parent1-class-name
{ definition code for parent2-class } ;
class grand-child-class-name : <access> parent1-class-name , <access> parent2-class-name { definition code for grand-child-class } ;

5) Example :-

```
class A//Grand Parent Class
{
 public:
 void display()
 {
  cout<<"\nClass A display() called";
 }
};
class B:virtual public A          //Parent1 class
{
 public:
 void display()
 {
  cout<<"\nClass B display() called";
 }
};
class C:virtual public A          //Parent2 class
{
 public:
 void display()
 {
  cout<<"\nClass C display() called";
 }
};
class D:public B, public C        //Grand Child Class
{
 public:
 void display()
 {
  cout<<"\nClass D display() called";
 }
};
```

```
main()
{
 clrscr();
 D ob;
 ob.display();
 ob.A::display();
 //Calling if A::display() could not have been done if
 //virtual was not used during inheritance
 ob.B::display();
 ob.C::display();
 getch();
}
```

12. Constructors and Destructors during Inheritance :-

1) When Inheritance takes place, an unnamed Instance of the Parent Class gets created for every Object created from the Child Class.
2) As an Instance of the Parent Class is created, so the constructor of the Parent Class also gets invoked.
3) If there is a Default Constructor in the Parent Class then it gets invoked First and then the Child Class constructor is invoked.
4) If there are only Parameterized Constructors in the Parent Class then we have to satisfy any of the Parameterized Constructors of the Parent Class through All the Constructors of the Child Class.
5) Only the Immediate Child Class Constructor can call the Immediate Parent Class Constructor to satisfy the compiler.
6) Only during Hybrid Inheritance and when we use Virtual Base class, then we need to call the Grand Parent class constructor through the Grand Child class constructor and also the usual calling of immediate parent class constructor by immediate child class constructor also needs to be followed. If Virtual Base class is not used then the grand child can not call the grand parent class constructor.
7) The Destructors are called in the just the Reverse order in which the constructors were called.
    i. Constructor of Parent → Constructor of Child.
    ii. Destructor of Child → Destructor of Parent.
8) Syntax :- child-class-constructor (arguments if any) : parent-class-constructor (parameters) { definition of the child-class-constructor }

9) Example :-

```
class A
{
 int a;
 public:
 A(int z)
 {
  cout<<"\nParameterized Constructor of Class A Called";
  a=z;
 }
 A(int u, int v)
 {
  cout<<"\nOverloaded Parameterized Contructor of Class A Called";
 }
 void show()
 {
  cout<<"\n"<<a;
 }
};
class B:public A
{
 int b;
 public:
 B(int z):A(z)                    //Parent class A constructor getting
invoked with parameter
 {
  cout<<"\nParameterized Constructor of Class B Called";
  b=z;
 }
 void show()
 {
  cout<<"\n"<<b;
 }
};
```

```
class C:public B
{
 int c;
 public:
 C():B(0)                    //Parent class A constructor getting invoked with
parameter
 {
  c=0;
  cout<<"\nDefault Constructor of Class C Called";
 }
 C(int z):B(z)               //Parent class A constructor getting invoked with
parameter
 {
  cout<<"\nParameterized Constructor of Class C Called";
  c=z;
 }
 void show()
 {
  cout<<"\n"<<c;
 }
};
main()
{
 clrscr();
 C ob;
 ob.A::show();
 ob.B::show();
 ob.show();
 C ob2(10);
 ob2.A::show();
 ob2.B::show();
 ob2.show();
 getch();
}
```

Pointer to Derived Classes :-

1. They are declared as pointers along with the class name as the data-type.
2. To access the members of the class arrow symbol "->" is used.
3. Its observed that the pointer of Parent class can point towards the objects of its Child class. But a Child class pointer cannot point towards Parent class.
4. Its observed that if pointer of Parent class is assigned address of its Child class then also its cannot access the members of the Child class object, although it can access the Parent class members.
5. Example :-

```
class A
{
 public:
 void show()
 {
  cout<<"\nWe are in Class A";
 }
};
class B:public A
{
 public:
 void show()
 {
  cout<<"\nWe are in Class B";
 }
};
main()
{
 clrscr();
 A oba;
 B obb;
 A *pa;
 pa=&oba;
 pa->show();               //This Calls Class A show()
 pa=&obb;
 pa->show();               //This also calls Class A show() Not Class B show()
 ((B*) pa)->show();
 //This calls Class B show() as pa is converted to B* type
 getch();
}
```

Virtual Functions :-

1. Virtual functions are defined using the keyword “virtual” before function declaration.
2. The access specifiers effect the accessibility of Virtual Functions.
3. Usually the Parent class pointer cannot access the member functions of its Child classes. Only when the prototype is same in both the Parent as well as Child classes and the function is declared as Virtual in the Parent class then it can be done.
4. If we desire to call the Child class function using the pointer of Parent class then :-
   1) In both the classes we have to define the function with same prototype.
   2) In the Parent class we need to use “virtual” keyword during defining the function.
   3) Now when we assign the address of Child class object to the pointer of the Parent class and using arrow operator we can call the function of the Child class.
5. Example :-

```
class A
{
 public:
 virtual void show()     //The show() function is declared Virtual
 {
  cout<<"\nWe are in Class A";
 }
};
class B:public A
{
 public:
 void show()
 {
  cout<<"\nWe are in Class B";
 }
};
main()
{
 clrscr();
 A oba;
 B obb;
 A *pa;
 pa=&oba;
 pa->show();   //This calls Class A show()
 pa=&obb;
 pa->show();   //This calls Class B show()
 getch();
}
```

Dynamic Binding or Runtime Polymorphism :-

1. It's the process of deciding at Runtime to either call the Parent class function or Child class function using Parent class Pointer.
2. Example :-

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
class A
{
 public:
 virtual void show()
 {
  cout<<"\nWe are in Class A";
 }
};
class B:public A
{
 public:
 void show()
 {
  cout<<"\nWe are in Class B";
 }
};

main()
{
 clrscr();
 A *pa;
 int inp;
 while(1)
 {
  cout<<"\nEnter 1 for Invoking Class A show()";
  cout<<"\nEnter 2 for Invoking Class B show()";
  cout<<"\nEnter 0 to EXIT\n\n";
  cin>>inp;
  switch(inp)
  {
   case 1:      pa=new A;
                //creating object of parent class A at Runtime and
                //assigning its address to pointer of parent class A
                pa->show();     //Invoking Class A's  show()
                break;
   case 2:      pa=new B;
                //creating object of child class B at Runtime and
                //assigning its address to pointer of parent class A
                pa->show();     //Invoking Class B's  show()
                break;
   case 0:      exit(0);
  }
 }
 getch();
}
```

Pure Virtual Functions and Abstract Classes :-

1. These are the functions that are declared as Virtual at the same time assigned to Zero value.
2. They don't have any definition, they are only declared and equated to Zero.
3. If a Class contains one or more Pure Virtual functions then that Class becomes Abstract.
4. Objects of Abstract classes can Not be created.
5. If we Inherit an Abstract class then we have to Redefine all the Parent class Pure Virtual functions in the Child class, otherwise the Child class would also become an Abstract class.
6. Syntax :- <u>virtual function-name (parameter list ,if any) = 0 ;</u>

7. Example :-

```
                class A         //It has become Abstract so No objects can be created from it
                {
                 virtual void show()=0;          //Pure Virtual Function
                };
                class B:public A
                {
                 public:
                            //Here we need to redefine show() otherwise class B also would
                            // become Abstract
                 void show()
                 {
                  cout<<"\nWe are in Class B";
                 }
                };
                main()
                {
                 clrscr();
                // A ob1;          //Error as A is Abstract, so class A objects can not be created
                 B ob2;
                 ob2.show();
                 getch();
}
```

CHAPTER 05 – Console, Disk I/O Operations and Templates

I/O Streams :-

1. A Stream is a queue of memory spaces or buffers that are used to connect I/O Devices and Disk Files with our programs.
2. We use Input Streams to connect our program with Input Devices or to Receive data from Disk Files.
3. We use Output Streams to connect our program with Output Devices or to Write data in to Disk Files.
4. All the I/O Streams are in the form of inbuilt Classes.
5. The Stream Class use for receiving input from any Input Device ISTREAM and from Disk File is IFSTREAM (which is derived form ISTREAM class).
6. The Stream Class use for sending output to any Output Device OSTREAM and to Disk File is OFSTREAM (which is derived form OSTREAM class).
7. All the Input and Output Stream Classes are Inherited from the grand parent IOS Class.
8. The IOSTREAM class for I/O Devices is derived from multiple inheritance from ISTREAM and OSTREAM classes.
9. The FSTREAM class for Disk Files is derived directly from IOSTREAM class.

Inbuilt Formatted Console Output functions :-

1. The inbuilt console Output operations can be used to set Format of how the output would appear.
2. All the console I/O functions are available in IOSTREAM class. Using “cin” object we can access the Input functions and using “cout” object we can access the Output functions.
3. Functions and their Use :-

Functions	***Use***
cout.setf(ios::left) ;	For displaying the matter using cout, as Left Justified
cout.setf(ios::right) ;	For displaying the matter using cout as Right Justified
cout.setf(ios::scientific) ;	For displaying the matter using cout in exponential (e) format.
cout.setf(ios::oct) ;	For displaying the matter in Octal format.
cout.setf(ios::hex) ;	For displaying the matter in Hexadecimal format.
cout . fill (char) ;	For displaying the specified characters instead of blank spaces.
cout . width (int) ;	Setting the number of columns in which the content has to be displyed. If the content length is less than the specified columns then the content will be justified according to the sef() function. If the columns specified is less than the matter length them the width is ignored and the full content is displayed. By default its left justified.
cout . precision (int) ;	It specifies the number of digits to be displayed after the decimal points.
cout . unsetf (float) ;	It resets the given “ ios : “ settings that are set using setf () function above it.

4. Example:-

```
main()
{
 clrscr();
 cout.fill('~');
 cout.width(30);
 cout.precision(3);
 cout<<123.1234;
// Output :-  ~~~~~~~~~~~~~~~~~~~~~~~123.123
 cout<<"\n";

 cout.setf(ios::showpos);
 cout.width(30);
 cout<<123.1234;
 cout.unsetf(ios::showpos);    // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~~+123.123
 cout<<"\n";

 cout.setf(ios::showpoint);
 cout.width(30);
 cout<<123;
 cout.unsetf(ios::showpoint);  // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~~~~~~~123
 cout<<"\n";

 cout.setf(ios::left);
 cout.width(30);
 cout<<123.1234;
 cout.unsetf(ios::left);             // Resetting the flag
//Output :-  123.123~~~~~~~~~~~~~~~~~~~~~~~
 cout<<"\n";

 cout.setf(ios::right);
 cout.width(30);
 cout<<123;
 cout.unsetf(ios::right);          // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~~~~~~~123
 cout<<"\n";
```

```
 cout.setf(ios::scientific, ios::floatfield);
 cout.width(30);
 cout<<123.1234;
 cout.unsetf(ios::scientific);     // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~1.231e+02
 cout<<"\n";

 cout.setf(ios::oct, ios::basefield);
 cout.width(30);
 cout<<10;
 cout.unsetf(ios::oct);  // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~12
 cout<<"\n";

 cout.setf(ios::hex, ios::basefield);
 cout.width(30);
 cout<<15;
 cout.unsetf(ios::hex);            // Resetting the flag
//Output :-  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~f
 cout<<"\n";

 cout<<15;
//Output :-  15
 getch();
}
```

Inbuilt Disk I/O Functions :-

1. The Disk I/O functions are needed to Read and Write into Disk files (Hard-disk).
2. The Inbuilt Disk I/O Functions are available in FSTREAM class, so we need to #include < fstream.h> header file.
3. Here the basic point of understanding of reading and writing should be only with respect to the Program we are writing.
4. If we say that we are Outputting data, then it means we are Writing data into a Disk File from our Program.
5. If we say that we are Inputting data, then it means we are Reading data from a Disk File to our Program.

6. Steps to be followed when using Disk files :-
   i) We create an fstream Object.
   ii) Using the fstream object now we open the specified File with specified Mode (Input, Output or Both).
   iii) Using the inbuilt member functions of the fstream object we perform the Reading or Writing operations.
   iv) Before finishing, we close the file and the stream through which the file is attached to our program, by using the close() function along with the fstream object.
7. Various Modes for Opening a File :-

Mode	***Meaning***
ios :: in	Opening the file in Input mode, for receiving data from the File. The File is NOT Created if it does not exist.
ios :: out	Opening the file in Output mode, for writing data into the File. Here the previous contents of the File are Deleted and new Data is written. The File is Created if it does not exist.
ios :: app	Opening the file in Append mode, for Writing data into the File. Here the Old contents remain intact and the new data is appended with previous contents. The File is Created if it does not exist.
ios :: nocreate	Opening fails if the Filename specified is not already present.
ios :: trunc	Delete all the contents of the File.
ios :: ate	Goto the end of file on opening.
Using piping operator | we can concatenate more than one mode at a time while opening a file. The combined effect of the specified modes would be seen when the file is opened. Ex:- F.open(“test.txt” , ios::in | ios::out) ; :- this would open the file in both read as well as write mode.	

8. Member functions of FSTREAM class and their Use :-

Functions	***Use***
open (name-of-file-in-double-quotes, mode)	For opening the specified File, whose name is given as a string in double quotes as the First argument and in the specified Mode given as the Second argument
close ()	Closes the file for stopping any further use.
put (char)	Writes one character at a time into the File, from the value of the char variable as the given argument.
get (char&) or int get()	Reads one character at a time from a File, into the char variable.
write ((char*) &variable , sizeof (variable))	Its used to write data of any data-type into the File, from the value of the variable given as the first argument.
read ((char*) &variable , sizeof (variable))	Its used to read data from the File into the variable given as the First argument.
eof(void)	Returns true (non zero value) when EOF is encountered.
fail()	Returns true (non zero value) when the file operation done immediately before, had Failed.

9. Following are some Pointers and Indicators associated with Files :-

   i) Beginning Of File (BOF) Indicator :- This indicator helps to identify the Beginning of File.
   Represented by :- ios :: beg

   ii) Current Position in File Indicator :- This indictor helps to identify the Current position where we are presently operating in the File.
   Represented by :- ios :: cur

   iii) End Of File (EOF) Indicator :- This indicator helps to identify the End of File.
   Represented by :- ios::end

   iv) Get Pointer :- The Get Pointer is used automatically when we are Reading contents from a File. The Get pointer gets incremented by the number of bytes automatically as soon as we read contents of the File. We can also manually set its position.
   Accessed by following functions :-
   - ✓ tellg(void) :- Gives the current position of the Get Pointer in the File.
   - ✓ seekg (moving-distance from starting-position , starting-position indicted by the indicator used) :- It moves the Get pointer to the distance indicated by the first argument from the reference position indicated by the second argument.

   v) Put Pointer :- The Put Pointer is used automatically when we are Writing data into a File. The Put pointer gets incremented by the number of bytes automatically as soon as we write data into the File. We can also manually set its position.
   - ✓ tellp(void) :- Gives the current position of the Put Pointer in the File.
   - ✓ seekp (moving-distance from starting-position , starting-position indicted by the indicator used) :- It moves the Put pointer to the distance indicated by the first argument from the reference position indicated by the second argument.

   vi) The Reading and Writing would take place from the Current Positions of Get and Put pointers respectively.

10. Example :-

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

class A
{
 int a;
 public:
 void fill()
 {
  cout<<"\nEnter Data :- ";
  cin>>a;
 }
 void show()
 {
  cout<<"\nValue of Data member is :- "<<a;
 }
};

main()
{
 clrscr();
 fstream f;
 int inpt;
 char cfile[]="c:\\CharFile.dat";
 char ofile[]="c:\\ObjFile.dat";
 while(1)
 {
  cout<<"\n\nEnter 1 for Writing Characters into the File (c:\\CharFile.dat)";
  cout<<"\nEnter 2 for Reading Characters from the File (c:\\CharFile.dat)";
  cout<<"\nEnter 3 for Writing Class Object into the File (c:\\ObjFile.dat)";
  cout<<"\nEnter 4 for Reading Class Object into the File (c:\\ObjFile.dat)";
  cout<<"\nEnter 5 for Deleting the File (c:\\CharFile.dat)";
  cout<<"\nEnter 6 for Deleting the File (c:\\ObjFile.dat)";
  cout<<"\nEnter 0 to EXIT\n";
  cin>>inpt;
```

```
switch(inpt)
{
 case 1:       f.open(cfile, ios::app);
               char dat1;
               cout<<"\nEnter a Character:- ";
               cin>>dat1;
               f.put(dat1);
               cout<<"\nPut pointer position:- "<<f.tellp();
               f.close();
               cout<<"\n-:OK:- ";
               break;
 case 2:     f.open(cfile, ios::in);
               if(f.fail())
               {
                cout<<"\nFile Read Error...";
                break;
               }
               char dat2;
               cout<<"\nData from File is...\n";
               while(!f.eof())
               {
                f.get(dat2);
                cout<<"\nGet pointer position:- "<<f.tellg();
                cout<<"   Data:- "<<dat2;
               }
               f.close();
               cout<<"\n-:OK:- ";
               break;
 case 3:       f.open(ofile, ios::app);
               A ob1;
               ob1.fill();
               f.write((char*)&ob1, sizeof(ob1));
               cout<<"\nPut pointer position:- "<<f.tellp();
               f.close();
               cout<<"\n-:OK:- ";
               break;
```

```
  case 4:     f.open(ofile, ios::in);
              if(f.fail())
              {
               cout<<"\nFile Read Error...";
               break;
              }
              A ob2;
              cout<<"\nData from File is...\n";
              while(!f.eof())
              {
               f.read((char*)&ob2, sizeof(ob2));
               cout<<"\nGet pointer position:- "<<f.tellg();
               if(f.eof()) break;
               ob2.show();
              }
              f.close();
              cout<<"\n-:OK:- ";
              break;
  case 5:     remove(cfile);
              cout<<"\nFile Deleted....";
              break;
  case 6:     remove(ofile);
              cout<<"\nFile Deleted....";
              break;
  case 0:     exit(0);
 }
}
getch();
}
```

Templates :-

1. There are two main categories of Templates :- 1) Function Templates and 2) Class Templates.
2. They are used for a Generic representation of Data-types , where multiple data-types can be implemented using the same definition of template.
3. It's a kind of Data-Type Polymorphism.
4. The Data-types need to be decided at compile time.
5. Syntax for Function Templates :-
   template <class T1, class T2, …., class Tn>
   return-type function-name(T1 a, …., Tn z)
   { function definition }
   Here we need to use the templates to be used in the input argument list
6. Syntax for Class Templates :-
   template <class T1, class T2, …., class Tn>
   class class-name
   { class definition }
   Here we need to use the templates to be used for variable declarations in the class.
   During creating Objects we need to specify the Data-Types along with the class name
   class-name<data-type1, data-type2, …, data-type n> object ;
7. Example :-

```
template <class T1, class T2>
class A
{
 T1 a;
 T2 b;
 public:
 void fill();
 void show()
 {
  cout<<"\n"<<a<<"\n"<<b;
 }
};
template <class T1, class T2>
void A<T1, T2>::fill()
{
  cin>>a>>b;
}
```

```
template <class T1>
class B
{
 T1 a;
 public:
 void fill()
 {
  cin>>a;
 }
 void show()
 {
  cout<<"\n"<<a;
 }
};
template <class T1>
void tfun1(T1 ob)
{
  ob.fill();
  ob.show();
}
template <class T1>
void tfun2(T1 z)
{
 cout<<"\n"<<z;
}
main()
{
 clrscr();
 A<char[100] , int> ob1;
 B<float> ob2;
 tfun1(ob1);
 tfun1(ob2);
 tfun2("Hello");
 tfun2(100);
 tfun2(10.5);
 getch();
}
```

Exception Handling :-

1. It's a procedure to ensure that during runtime if a specific error is encountered then this facility would help to handle the condition and necessary actions can be taken, rather than the system terminating our program abruptly.
2. For exception handling we use " try-throw-catch " statement.
3. Using try-catch block either a specific exception can be handled or all exceptions can be handled.
4. The program code that can possibly generate exception is put inside " try " block. When an exception condition occurs then " throw " clause is used to throw an exception. The " catch " block is used to catch the thrown exceptions.
5. Syntax for handling specific exceptions :-

```
try
{
            // Exception condition occurred so we use throw clause.
    throw value1;
            // Exception condition occurred so we use throw clause.
    throw value2;
}
            // The catch block having the same data type as that of the throw clause will
            // be used for handling the exception
catch ( datatype1 var )
{
    statements to handle the exception situation.
}
catch( datatype2 var )
{
    statements to handle the exception situation.
}
```

6. Syntax for handling all exceptions :-

```
try
{
    Exception condition occurred so we use throw clause.
    throw value;
}
catch (. . .)
{
    statements to handle the exception situation.
}
```

7. Example :-

```
main()
{
	try
	{
		cout<<"\nEnter a Number:- ";
		int val1;
		cin>>val1;
		if(val1==1) throw val1;		// It would be caught by catch(int a)
		cout<<"\nEnter a Character:- ";
		char val2;
		cin>>val2;
		if(val2=='Z') throw val2;		// It would be caught by catch(char c)
		throw 1.5;				// It would be caught by catch(…)
	}
	catch(int a)
	{
		cout<<"\nException Caught:- "<<a<<"\n";
	}
	catch(char c)
	{
		cout<<"\nException Caught:- "<<c<<"\n";
	}
	catch(...)
	{
		cout<<"Generic Exception Caught";
	}
}
```

BEST OF LUCK