OBJECTED ORIENTED PROGRAMMING WITH C++

UNIT I

Introduction: A Look at procedure oriented programming-Object Oriented Programming Paradigm-Basic concepts of Object Oriented Programming-Benefits of OOP-Application of OOP.

Simple I/O Operation: cin and cout statements.

UNIT II

Classes and Objects: Structure of C++ Program-Specifying a class-Defining member functions-Static data members and static member functions-Object as function arguments-Friendly functions.

UNIT III

Constructors and Destructors: Constructors-Parameterized constructors-Multiple constructors in class-Destructors.

Operator overloading: Defining operator overloading-overloading unary operators-overloading binary operators, overloading binary operators using friends-Rules for operator overloading-Type conversion.

UNIT IV

Inheritance: Defining derived classes-Single Inheritance-Multilevel inheritance-Multiple inheritance.

Pointers: Declaring and initializing pointers-pointers expressions and pointer arithmetic – using pointers with arrays and strings-this pointer-pointers to derived classes.

UNIT V

Virtual functions and Pure Virtual functions.

Working with Files: Introduction-Classes for file stream operators-Opening and Closing a File - checking for eof() - File Modes-get () and put () functions- Binary files - read and functions-reading and writing a class object.

Reference Book: Objected Oriented Programming with C++, E.Balagurusamy, 6 edition, McGraw Hill Education (India) Private Limited, New Delhi.

UNIT I

1.1 INTRODUCTION: A LOOK AT PROCEDURE ORIENTED PROGRAMMING

Procedure Oriented Programming(POP) vs Object Oriented Programming(OOP)

Procedure Oriented Programming :-

Procedure Oriented Programming (POP) approaches the problem as a sequence of things to be done such as reading, calculating and printing. The primary focus is on Functions. Procedure oriented programming basically consists of writing a set of instructions for the computer to follow, and organizing these instructions into groups are known as functions. Functions are defined to accomplish the tasks to be carried out. (Figure 1.1)

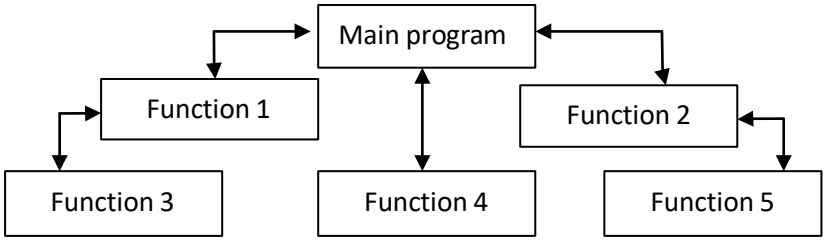


Fig. 1.1 Structure of procedure oriented programs

Characteristics of Procedure Oriented Programming:

- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Employs top-down approach in program design.
- Importance is not given to **data** but to functions as well as **sequence** of actions to be done.
- Does not have any access specifier.
- Data can move freely from function to function in the system.
- Adding new data and function is difficult.
- Most function uses Global data for sharing that can be accessed freely from function to function.
- POP does not have any proper way for hiding data so it is **less secure**.
- Overloading is not possible.
- Examples: C, VB, FORTRAN, Pascal.

Object-Oriented Problem Solving Approach

It consists of identifying *objects* and how to use these *objects* in the correct sequence to solve the problem. Object-oriented problem solving consist of designing objects that helps to solve a specific problem. A message to an object causes it to perform its operations and solve its part of the problem.

The object-oriented problem solving approach, in general, can be divided into following four steps.

- Identify the problem
- Identify the objects needed for the solution
- Identify messages to be sent to the objects
- Create a sequence of messages to the objects that solve the problem.

1.2 OBJECT ORIENTED PROGRAMMING PARADIGM

Object oriented programming allows a decomposition of a problem into a number entities called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Sequential Operation

In a sequential operation the messages are sent to objects in sequential order. Control will not return to the original sending object until all other messages have been completed. For example, in the following diagram (Figure 1.2). Object A sends a message to Object B which in turn sends a message to Object C. Object C has to return to Object B which then returns to Object A. Control does not return to Object A, until all the other messages have completed.

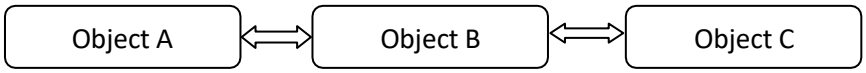


Fig 1.2 Sequential operations in objects

In object-oriented programming, a method is a programmed procedure that is defined as part of a class and included in any object of that class. A class (and thus an object) can have more than one method.

Data is represented as properties of the object and behavior as methods**.** When an object receives a message, it determines what method is being requested and passes control to the *method*. An object has as many methods as it takes to perform its designed actions.

Method – A method (or message) in object-oriented programming (OOP) is a procedure associated with an object. An object is made up of data and behavior, which form the interface that an object presents to the outside world.

Methods that operate on specific objects are *instance methods* and messages that invoke instance methods are called *instance message*. Methods that operate on specific classes are *class methods*.

Desirable qualities of an object are,

Modularity: The operations / methods should be defined separately for each module. One of the objects‟ features can be used by other object as a reusable component.

Information Hiding: The data about one object should exclusively belong to that object, which enables the individual modules focuses on specific functionality.

Re-usability of code: An object of one class should have the same functionality that of another. Same code could be usable by all objects of one class.

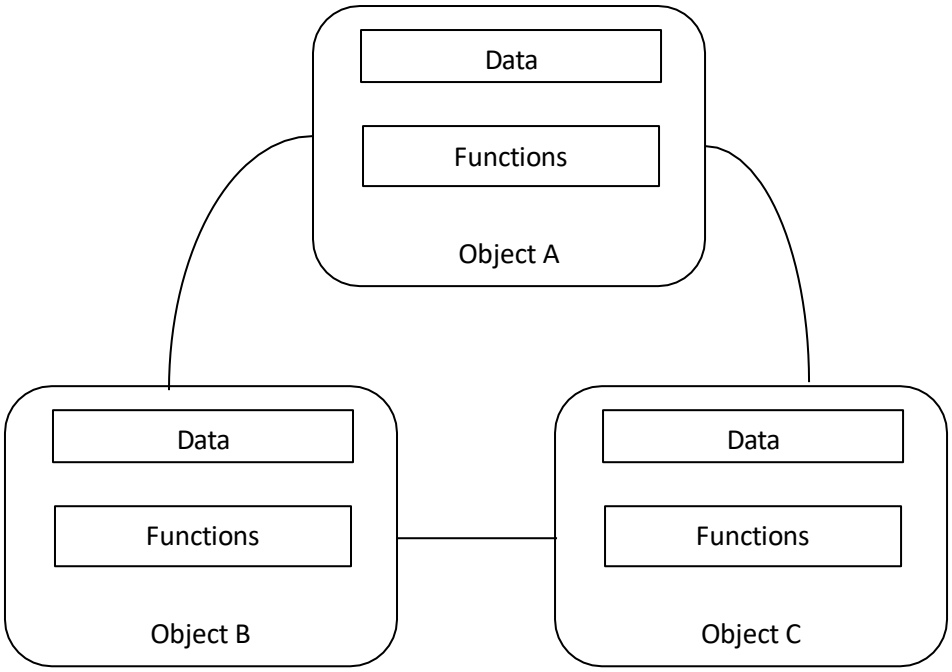


Fig 1.3 Object representation with data and functions

The object oriented programming can be defined as an approach that modularizes programs by creating partitioned memory area for both data and functions (Figure 1.3). The modules can be used as templates for creating other modules if needed. Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Some characteristics of Object Oriented Programming are :-

- Programs are modularized objects.
- Functions that operate on the data are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through member functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.
- Has access specifiers named Public, Private, Protected, etc.
- Provides an easy way to add new data and function
- Data Hiding provides **more security**.
- Overloading is possible in the form of Function Overloading and Operator Overloading.
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object‟s data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes
- Examples: C++, Java, Smalltalk, Delphi, C#, Perl, Python .NET, PHP.

1.3 BASIC CONCEPTS OF OBJECT ORIENTED PROGRAMMING

The objected oriented paradigm is not just a programming style but also a design method for building systems. Defining an object in building a system is more important.

Object

In the class-based object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.

- Object contains data and the code to manipulate that data.
- An object can be considered a "thing" that can perform a set of activities that defines the object's behaviour or characteristics.
- The object's interface consists of a set of commands, each command performing a specific action.
- An object asks another object to perform an action by sending it a *message*. The requesting (sending) object is referred to as *sender* and the receiving object is referred to as *receiver*.
- Control is given to the receiving object until it completes the command; control then returns to the sending object.
- A message can also contain information the sending objects needs to pass to the reviewing object, called the *argument* in the message.
- A receiving object always returns a *value* back to the sending object. This returned value may or may not be useful to the sending object.

Classes

- Classes are user defined data type.
- Object can be made as a user defined data type with the support of a class.
- Objects are the variables of the data type class.
- A created class can contain any number of objects.
- A class is a collection of objects of similar type.
- Each object is connected with the class with which it was created.
- Syntax for creating an object from class is

Classname Objectname

Example: Car, Bus, Motor bike are members of the class Vehicle. If Vehicle is defined as a class, then objects can be created as follows:

Vehicle Car

Object oriented languages allow the programmer to specify self contained units and specify them as objects which are made up of data and the methods or operations which can be performed on the object. A computer language is object-oriented if they support the following properties:

Encapsulation

- Encapsulation is the process of hiding data of a class from objects.
- Encapsulation bundles the actions and attributes together as a single unit.
- The advantage of encapsulation is that the implementation is not accessible to the client.
- The user knows only the functionality of encapsulated unit and information to be supplied to get the result.
- Encapsulation supports ***information hiding*** by making use of the three access specifiers of a class.

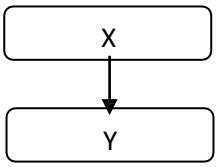
Data Hiding / Information hiding

- The process of insulating an object‟s data is called data hiding or information hiding.
- Isolates the end users from the requirement of intimating knowledge of the design for the usage of a module.
- Information hiding access specifiers are,
    - Public: If a class member is public, it can be used anywhere without the access restrictions.
    - Private: if a class member is private, it can be used only by the members and friends of class.
    - Protected: if a class member is protected, it can be used only by the members and friends of class and the members and friends of classes derived from class.

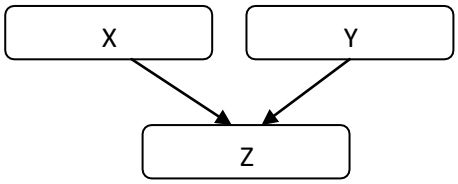
Inheritance

- Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities.
- Inheritance defines an “is – a” relationship.
- The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses.
- The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so.
- Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods.

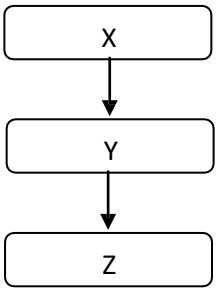
- Types of Inheritance:
  - **Single Inheritance** : A subclass derives from a single super-class.



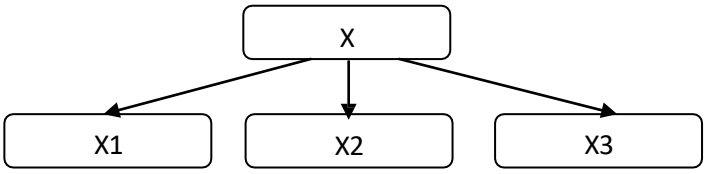
  - **Multiple Inheritance** : A subclass derives from more than one super-classes.



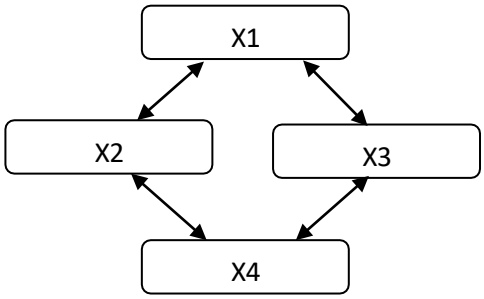
  - **Multilevel Inheritance** : A subclass derives from a super-class which in turn is derived from another class and so on.



  - **Hierarchical Inheritance** : A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.



  - **Hybrid Inheritance** : A combination of multiple and multilevel inheritance so as to form a lattice structure.



Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

Polymorphism

- Polymorphism is originally a Greek word that means the ability to take multiple forms.
- It implies using operations in different ways, depending upon the instance they are operating upon.
- Polymorphism allows objects with different internal structures to have a common external interface.

Generalization

- In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy,
- Subclasses are combined to form a generalized super-class.
- It represents an "is – a – kind – of" relationship.

Specialization

- Specialization is the reverse process of generalization.
- Distinguishing features of groups of objects are used to form specialized classes from existing classes.
- The subclasses are the specialized versions of the super-class.

Message Passing

- Objects in a system may communicate with each other using message passing.
- The features of message passing are:
    - Message passing between two objects is generally unidirectional.
    - Message passing enables all interactions between objects.
    - Message passing essentially involves invoking class methods.

Link

- A link depicts the relationship between two or more objects.
- A link represents a connection through which an object collaborates with other objects.

- Through a link, one object may invoke the methods or navigate through another object.

Association

- Association is a group of links having common structure and common behavior.
- Association depicts the relationship between objects of one or more classes.

Degree of an Association

- Degree of an association denotes the number of classes involved in a connection.
- Degree may be unary, binary, or ternary.
    - A **unary relationship** connects objects of the same class.
    - A **binary relationship** connects objects of two classes.
    - A **ternary relationship** connects objects of three or more classes.

1.4 BENEFITS OF OBJECT ORIENTED PROGRAMMING

The benefits of using the object model are:

- It helps in faster development of software.
- Inheritance eliminates redundant code and extends the use of existing classes which is not possible in procedure oriented approach.
- Building programs from the standard working modules that communicate with one another thereby saving of development time and higher productivity.
- Information /Data hiding builds a secure system.
- Multiple instances of object can co-exist without any interference.
- Partitioning the project work based on objects.
- Can be easily upgraded from small to large systems.
- Message passing techniques makes the interface descriptions with external systems easier and simpler.
- It is easy to maintain.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

1.5 APPLICATIONS OF OOP

- ***Real time Systems*** - A real time system is a system that give output at given instant and its parameters changes dynamically. Code changing is very easy in OOP system

and it leads toward dynamic behaviour of OOP codes making it more suitable to real time system.

- ***Simulation and Modelling*** - in the area of System modelling where criteria for OOP approach is used. Representing a system is very easy in OOP approach
- ***Hypertext And Hypermedia*** - Hypertext and hypermedia is another area where OOP approach is used. Its ease of using OOP codes that makes it suitable for various media approaches.
- ***Decision support system*** - Decision support system is an advance and complex Real time system where OOPS can be applied.
- ***CAM/CAE/CAD System*** - Computer has wide use of OOP approach due to time saving in writing OOP codes and dynamic behaviour of OOP codes.
- ***Office Automation System*** - Embeded systems make it easy to use OOP for automated system.
- ***AI and expert system*** - It is mixed system having both hypermedia and real time system.
- ***Real-time Business System-*** OOP has gained importance in the area of on-line business via Internet.

The various differences between the procedure oriented programming and Object Oriented programming are listed in Table 1.1.

Table 1.1 Difference between Procedure Oriented programming: **and OOP**

POP	OOP
follows a top down approach	follows a bottom up approach
Importance is given to the sequence of things to be done	Importance is given to the data
larger programs are divided into functions	larger programs are divided into objects
most functions share global data.	mostly the data is private
importance to algorithm rather than data	importance to data rather than algorithm
Example: Pascal and C	Example: C++ and Java

1.6 SIMPLE I/O OPERATION: CIN AND COUT STATEMENTS.

1.6.1 C++ BASICS

- **Header files** are included at the beginning similar to C program. Header files contained pre-declared function libraries, which can be used by users for their ease. ***iostream*** is a header file which provides with input & output streams. It is represented as

#include <iostream.h>

- ***main()***, is the function which holds the executing part of program. The return type ***main*** function is ***int.***
- ***cout*** <<, is used to print anything on screen
- **Comments –**
    - For single line comments, // is used before mentioning comment. For example, cout<<"hello "; // This is single line comment
    - For multiple line comment, enclose the comment between /* and */. For example,

```
/*this is a multiple line comment
……
….
Comment ends */
```

1.6.2 Data Types in C++

Data types can be either built in or abstract data type.

Basic Built in types

Type	Size	Purpose	Example
char	1 byte	For character	char a = 'A';
Int	2 bytes	For integral number	int a = 1;
float	4 bytes	For single precision floating point	float a = 3.14159;
double	8 bytes	For double precision floating point numbers	double a = $6e^{-4}$;

Types of operators

- ***Assignment Operator***

- Operator '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

- ***Mathematical Operators***
  - There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.
- ***Relational Operators***
  - These operators establish a relationship between operands. The relational operators are : less than (<) , grater than (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).
- ***Logical Operators***
  - The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.
- ***Bitwise Operators***
  - There are used to change individual bits into a number. They work with only integral data types like char, int and long and not with floating point values.
    - Bitwise AND operator - &
    - Bitwise OR operator - |
    - Bitwise XOR operator - ^
    - Bitwise NOT operator - ~
- ***Shift Operators***
  - Shift Operators are used to shift Bits of any variable. It is of three types,
    - Left Shift Operator - <<
    - Right Shift Operator - >>
    - Unsigned Right Shift Operator - >>>
- ***Unary Operators***
  - These are the operators which work on only one operand. There are many unary operators, but increment++ and decrement -- operators are most used.

- **Other Unary Operators :** address of &, dereference *, **new** and **delete**, bitwise not ~, logical not !, unary minus - and unary plus +.

- ***Ternary Operator***
  - The ternary if-else? : is an operator which has three operands. For example,

```
int x = 100;
x > 50 ? cout << "true" : cout << "false"
```

- ***Comma Operator***
  - This is used to separate variable names and to separate expressions. st of memory allocated for data types, objects and user defined data types.

* * *

UNIT II

2.1 STRUCTURE OF C++ PROGRAM

C++ program has four important sections namely,

- Header files to be included
- Class declarations
- Member function definitions
- Main function

The sections can be placed as different code files, which can be compiled together or independently.

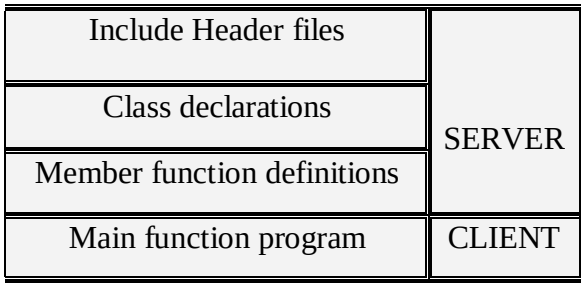


Fig 2.1 Structure of C++ program

C++ structure is organized as illustrated in the Figure 2.1. Since class declarations and member function definition are separated, it helps the programmer to separate the abstract specification of the interface from the implementation details.

The Main function includes all the required files in all the sections for execution. The class definitions and the member functions serve as the server to provide services to the client i.e the main program. Public Interface of the class is used for communication between client and server elements of the program.

2.2 SPECIFYING A CLASS

Class can be considered as a blueprint or a template. A class binds the data and its associated functions. Specification of a class has two parts namely,

- class declaration – describes the type and scope its members.
- class function definition – describes the implementation of class function.

- Class members are the variables and functions declared inside the class. The keywords private and public are called as visibility labels.
- Variables inside the class are called as data members and the functions are known as member functions.

Syntax of class declaration

```
class class_name
{   private :
        variable declarations;
        function declarations;
public :
        variable declarations;
        function declarations;
};
```

- Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
- Classes contain data members and member functions, and the access of these data members and variable depends on the access specifiers.
- Class's member functions can be defined inside the class definition or outside the class definition.
- By default, a class has private access control.
- No storage is assigned during a class definition.
- Class name must start with an uppercase letter.

Access Control in Classes

- Access specifiers are used to set boundaries for availability of members of class.
- Access specifiers in the program, are followed by a colon.
- Either one, two or all 3 specifiers can be used in the same class to set different boundaries for different class members.
- Access specifiers in C++ class are public, private and protected
    - **Public -** All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too.

```
class PubAccess
{
 public:   // public access specifier
 int a;          // Data Member Declaration
```

```
 void fun1( );  // Member Function
declaration
};
```

- **Private - N**o one can access the class members declared private outside that class. If tried to access the private member, a compile time error will occur. By default class variables and member functions are private.

```
class PriAccess
{
 private:   // private access specifier
 int a;          // Data Member Declaration
void func2( );  // Member Function
declaration
};
```

- **Protected -** Class member outside the class are inaccessible but they can be accessed by any subclass of that class. If class A is inherited by class B, then class B which is subclass of class A can access.

```
 class ProAccess
{
 protected:  // protected access specifier
int a;          // Data Member Declaration
void show();  // Member Function
declaration
};
```

Creating objects: Once class has been defined any number of objects belonging to that class can be created. Objects are the variable of the type class. A class is a collection of objects of similar type. The declaration of object allocates needed memory space for the object. For example the following class has three objects namely obj1, obj2 and obj3.

```
class cname
{
```

```
…
…
}obj1, obj2, obj3;
```

2.3 DEFINING MEMBER FUNCTIONS

- Class is a user defined data type, that holds its own data members and member functions, which can be accessed and used by creating instance of that class.
- The variables inside class definition are called as data members and the functions are called member functions.
- Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.
- If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

Example :

```
class BoxVolume
{
 public:
 int a;
 int getVolume();     // Declaring
function getVolume with no
argument and return type int.
};
```

If function is defined inside class then declaring it first is not needed. The function can directly be defined.

```
class BoxVolume
{
 public:
 int a;
 int getVolume()
 {
```

```
    return a*a*a;
   //returns volume of cube
    }
   };
```

But if the member function is defined outside the class definition then the function must be declared inside class definition and then define it outside.

```
class BoxVolume
{
 public:
 int a;
 int getVolume();
}

int BoxVolume :: getVolume()    //
defined outside class definition
{
 return a*a*a;
}
```

Types of Member Functions

Following are different types of Member functions,

- Simple functions
- Static functions
- Const functions
- Inline functions
- Friend functions

Simple Member functions

- These are the basic member function, which do not have any special keyword prefix.
- The syntax is as shown below:

```
return_type
functionName(parameter_list)
{
```

```
function body;
}
```

Static Member functions

- Static holds its position.
- Static is a keyword which can be used with data members as well as the member functions.
- A function is made static by using static keyword with function name. These functions work for the entire class rather than for a particular object.

Example:

```
class A
{
 public:
 static void f() { };
};
int main()
{
 A::f( );  // calling member function directly with
class name
}
```

- It can be called using the object and the direct member access . operator.
- Calling using class name and scope resolution :: operator is also allowed.
- These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

Const Member functions

- Const keyword makes variables constant, whose values remains constant and cannot be changed.
- When used with member function, the member functions can never modify the object or its related data members.
- Basic Syntax of const Member Function is

```
void func( ) const { }
```

Inline functions

- All the member functions defined inside the class definition are by default declared as Inline.

Friend functions

- Friend functions can have a **private** access to non-class functions.
- A global function or a member function of other class can be declared as friend.
- Friend functions are actually not class member function.
- Friend functions can access private data members by creating object of the class.
- When a class is made as a friend, all its member functions automatically become friend functions.

Example :

```
class MyFriend
{
 int i;
 public:
 friend void func( );  // Global function as friend
};
void func( )
{
 MyFriend wf;
 mf.i=100;  // Access to private data member
 cout << mf.i;
};
int main( )
{
func( ); // Called directly
}
```

2.4 STATIC DATA MEMBERS AND STATIC MEMBERS FUNCTIONS

Characteristics of Static data member in class

- Static member variable is initialized to zero when the first object of its class is created.
- Once the definition for static data member is made, user cannot redefine it.

- Static data members of class are the members that are shared by all the objects of that class.(Fig. 2.2)

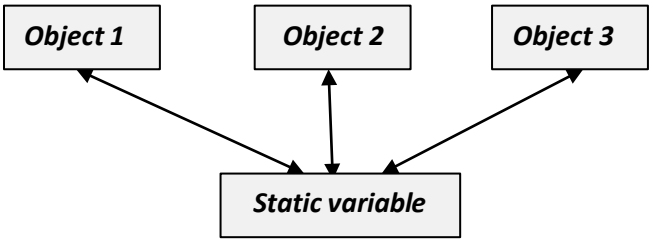


Fig. 2.2.Sharing of static data member

- It is visible only within the class.
- Static data member has a single piece of storage.
- Static member variables (data members) are not initialised using constructor, because these are not dependent on object initialization.
- Used to maintain values common to the entire class.
- It must be initialized explicitly, always outside the class. If not initialized, a Link error will occur

```
class A
{
 static int i;
 public:A( )
{       };
};
int A::i=1;
int main()
{
 A obj;
 cout << obj.i;   // prints value of i
}
```

Static Member Functions

Characteristics of static member function are,

- Can have access only to other static members declared in the same class.

- These functions or variables cannot access ordinary data members and member functions.
- These functions work for the class as a whole rather than for a particular object of a class.
- A static member function can be called using an object and the direct member access . operator.
- It can be called by itself, using class name and scope resolution :: operator.
- Syntax – ***class_name :: function_name***

Example :

```
class A
{
 public:
 static void f( )  { };
};
int main()
{
A::f( );  // calling member function directly with class
name
}
```

2.5 OBJECT AS FUNCTION ARGUMENTS

An object can be used as a function argument using the following two methods:

- ***Pass by value*** :Passing a copy of the entire object to the function. Changes made to the object inside the function do not affect the object used to call the function.
- ***Pass by reference:*** Transferring the address of the object alone to the function. The called function works directly on the actual called object. Changes made to the object inside the function affects the object used to call the function.

Characteristics:

- Every call to a member function is associated with a particular object.
- Using the member names alone the function has direct access to all the members, whether private or public, of that object.

- It also has indirect access, using the object name and the member name, connected with the dot operator to other objects of the same class that are passed as arguments.
- An object can also be passed as an argument to a non-member function. The non-member function can have access to the public member functions only through that objects. These non-member functions cannot have access to the private data members.
- Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

2.6 FRIENDLY FUNCTIONS

In C++, a non-member function cannot have an access to the private data of a class. Normally, a **function** that is defined outside of a class cannot access such information. A **friend function** of a given class is allowed access to private and protected data in that class.

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- The functions that are declared with the keyword ***friend*** are known as friend functions.
- A friend can be one of the following in case the entire class and all of its members are friends.
    - a function
    - a function template
    - a member function
    - a class
    - a class template
- To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:

```
class class_name
{
   ..... .... ........
   friend return_type function_name(arguments);
   ..... .... ........
};
```

- The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.
- It can be declared either in the public or private part of the class.
- A function can be declared as a friend in any number of classes.
- If a function is defined as a friend function then, the private and protected data of class can be accessed from that function. The complier knows a given function is a friend function by its keyword friend.
- A friend function has full access rights to the private members of the class.
- A class can be made a friend of another class using keyword ***friend***. For example:

```
........ ..... ........
class MainA
{
  friend class SubB;      // class B is a friend class
  ..... ..... .....
};
class SubB
{
  ..... ..... .....
};
```

- When a class is made a ***friend class***, all the member functions of that class becomes friend function.

- A friend function can be invoked like a normal function without the help of any object.
- It is not in the scope of the class to which it has been declared as friend and hence, it cannot be called using the object of that class.
- A member function of one class can be friend function of another class.
- A friend function can be called by reference.

UNIT III

3.1 CONSTRUCTORS AND DESTRUCTORS

Constructors

- Constructors are class functions that perform initialization of every object.
- Constructors initialize values to object members after storage is allocated to the object.
- The Compiler calls the Constructor whenever an object is created.
- The name of constructor will be same as the name of the class

  Example:

```
class X
{
 int i;
 public:
 X( );  //Constructor
};
```

- Constructors will not have a return type.
- Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class X
{
 int i;
 public:
 X( ); //Constructor declared
};

X::X( )   // Constructor definition
{
 i=10;
}
```

Types of Constructors

The three types of Constructors are:

- Default Constructor
- Parameterized Constructor

- Copy Constructor

Default Constructor

- Default constructor does not take any argument.
- It has no parameter.

Syntax :

```
class_name ()
{
 Constructor Definition
}
```

Example :

```
class Cube
  {
  int  side;
   public:
   Cube()
          {
            side=10;
             }
       };

int main()
{
        Cube c;
         cout << c.side;
}

Output : 10
```

After creating the object, the constructor is called which initializes its data members. A default constructor is used for initialization of object members

Even if a constructor is defined explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
  {
    int side;
  };

  int main()
    {

      Cube c;
       cout << c.side;

        }
Output : 0
```

In the above example, the default constructor provided by the compiler is called which has initialized the object data members to default value of 0.

Characteristics of Constructors

- Called automatically when the objects are created.
- All objects of the class having a constructor are initialized before using.
- Declared in the public section for availability to all the functions.
- Default and copy constructors are generated by the compiler wherever required.
- Generated constructors are public.
- Can have default arguments like other C++ functions.
- A constructor can call member functions of its class.
- Return type **cannot** be specified for constructors.
- Constructors **cannot** be inherited, but a derived class can call the base class constructor.
- Constructors **cannot** be static.
- An object of a class with a constructor **cannot** be used as a member of a **union.**

3.2 PARAMETERIZED CONSTRUCTORS

- The constructors possess parameters.
- Different values to data members of different objects can be provided, by passing the appropriate values as argument.

Example :

```
class Cube
{
 int side;
public:  Cube(int x)
   {
     side=x;
    }
};
 int main()
{
Cube cb1(100);
Cube cb2(200);
Cube cb3(300);
cout << cb1.side;
cout << cb2.side;
cout << cb3.side;
}

OUTPUT : 100 200 300
```

In the above example, the parameterized constructor has initialized three objects ***cb1, cb2*** and ***cb3*** with user defined values as 100, 200 and 300 respectively.

3.3 MULTIPLE CONSTRUCTORS IN CLASS

- A class can contain more than one Constructor. This is called as Constructor Overloading.
- All constructors are defined with the same name as that of the class they belong to.
- All constructors can contain different number of arguments.
- Based on the number of arguments, the compiler executes the appropriate constructor.

Example:

```
        // constructor declaration
cons(int n, float m, char p);
cons(int n, float m);
cons( );
            // object declaration
cons A(10,20,25,"S" );
cons B( 100, 1.2);
cons C;
```

In the above example the compiler decides which constructor to be called based on the number of arguments in the object. When A is created, the constructor with three arguments is called since the declaration of the object is followed with three arguments. Similarly, for object A and object C the constructor with two arguments and one argument is called respectively.

3.4 DESTRUCTORS

- Destructor is a class function that destroys the object as soon as the scope of object ends.
- The destructor is called automatically by the compiler when the object goes out of scope.
- In the syntax of destructor, the class name is used for the name of destructor; with a **tilde** ~ sign as prefix to it.
- Destructors will never have any arguments.

Characteristics

- It does not take any parameter nor does it return any value.
- Overloading a destructor is not possible.
- A class can have only one destructor. A destructor can be defined outside the class.
- Called automatically when the objects are destroyed.
- Destructor functions follow the usual access rules as other member functions.
- De-initializes each object before the object goes out of scope.
- Cannot be inherited.
- Address of a destructor cannot be taken.

- A destructor can call member functions of its class.
- An object of a class having a destructor cannot be a member of a union.

```
class X
{
    public:
    ~X( );
};
```

Example: Calling Constructor and Destructor

```
class X
{
X( )
 {
  cout << "Called Constructor";
 }

~X( )
 {
  cout << "Called Destructor";
 }
};

int main()
{
X obj1;  // Called  Constructor
 int i=1;
 if(i)
  {
   X  obj2;  // Called Constructor
  }   // Destructor Called for obj2
} //  Destructor called for obj1
```

3.5 OPERATOR OVERLOADING – DEFINING OPERATOR OVERLOADING

- Operator Overloading provides new function definitions to the normal C++ operators like +, -, *, ++, --, +=,-=, <, >. The mechanism of providing such an additional definition to an operator is known as operator overloading in C++.
- The overloaded function definitions are for user defined datatypes.
- Existing operators alone can be overloaded.

- For example, the functionality of „+“ operator can be extended to strings through operator overloading instead of using ***strcat()*** function.

Syntax:

```
Returntype classname :: operator operatorsymbol(argument list)
{
// function definition
}
```

Steps for creating overloaded operators:

a. Create a class that defines the data type for overloading operation.
b. Declare the operator function in the public part of the class
c. Define the operator function for the needed operator.

Operator overloading can be done by implementing a function which can be :

- ***Member Function*** - Operator overloading function can be a member function if the Left operand is an Object of that class.
- ***Non-Member Function*** - if the Left operand is different, then Operator overloading function must be a non-member function
- ***Friend Function*** - Operator overloading function can be made friend function if it needs access to the private and protected members of class.

The Syntax of declaration of an Operator function is as follows:

```
operator Operator_name
```

Example – declaring an Operator function for „=“ is, ***Operator =***

3.6 OVERLOADING UNARY OPERATORS

Unary operators act on only one operand where an operand is simply a variable acted on by an operator. There is no restriction on the return types of the unary operators. The unary operators that can be overloaded are the following:

- **!** (logical NOT)
- **&** (address-of)
- **~** (one's complement)
- ***** (pointer dereference)

- + (unary plus)
- - (unary negation)
- ++ (increment)
- -- (decrement)

- To declare a unary operator function as a non-static member, declaration syntax is of form:

```
ret_type operatorop( )
```

where

ret-type - the return type, *op* - one of the operators.

- To declare a unary operator function as a **global function**, declaration syntax is of form:

```
ret_type operatorop(arg )
```

where *arg* is an argument of class type on which to operate.

- The keyword ***operator*** is used to overload the ++ operator. Example:

```
void operator ++ ( )
```

The return type void comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses. This declaration syntax invokes the compiler to call this member function whenever the ++ operator is encountered.

```
++c1;  //  increment count

C1 // object
…
…
count  //function
…
…
void operator++( )  // no arguments
{
   ++count;
}
```

- Whenever an unary operator is used, it works with one operand. Therefore with the user defined data types, the operand becomes the caller and hence no arguments are required.

Postfix form

- When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type **int.**
- Specifying any other type generates an error.
- The argument of type **int** that denotes the postfix form of the increment or decrement operator is not commonly used to pass arguments. It usually contains the value 0.

The operator functions namely operator=, operator [ ], operator ( ) and operator? are non-static member functions.

Some examples of declarations of operator functions are given below:

```
class X
{
X operator ++ (int);//Postfix increment
X operator ++ ( ); //Prefix increment
}
```

3.7 OVERLOADING BINARY OPERATORS

- A Binary Operator can be defined either a member function taking one argument.
- An operator function should be either a member or take at least one class object argument.
- An operator function, which needs to accept a basic type as its first argument, cannot be a member function.
- All function work with two operands. The first (Rational) is the operator overloaded function caller and the second (object) is the passed argument.
- The following is the list of binary operators that can be overloaded:

Operator	Name	Operator	Name
,	Comma	*	Multiplication
!=	Inequality	*=	Multiplication/assignment
%	Modulus	+	Addition

%=	Modulus/assignment	+=	Addition/assignment
&	Bitwise AND	–	Subtraction
&&	Logical AND	–=	Subtraction/assignment
&=	Bitwise AND/assignment	–>	Member selection
/	Division	–>*	Pointer-to-member
/=	Division/assignment	^	Exclusive OR
<	Less than	^=	Exclusive OR/assignment
<<	Left shift	|	Bitwise inclusive OR
<<=	Left shift/assignment	|=	Bitwise inclusive
<=	Less than or equal to	||	Logical OR
=	Assignment	>>	Right shift
==	Equality	>>=	Right shift/assignment
>	Greater than	>=	Greater than or equal to

- To declare a binary operator function as a nonstatic member, its declaration syntax is of the form:

```
ret_type operatorop(arg )
```

where *ret-type* is the return type, *op* is one of the operators listed in the table, and *arg* is an argument of any type.

- To declare a binary operator function as a global function, its declaration syntax is of the form:

```
ret_type operatorop(arg1,arg2 )
```

where *arg1* and *arg2* are arguments. At least one of the arguments must be of class type.

- There is no restriction on the return types of the binary operators.
- Normally, user-defined binary operators return either a class type or a reference to a class type.

3.8 OVERLOADING BINARY OPERATORS USING FRIENDS

- Friend function can be used in the place of member functions for overloading a binary operator.
- A friend function is in need of two arguments to be explicitly passed to it.
- In situations where we need to use two different types of operands for a binary operator friend function is more useful than member function. One operand can be an object and another can be a built-in data type

Overloading the operators >> and << using friend function

In the statement cin >> a; the right shift operator >> takes two operands, one is an object of **istream** and another is an **integer** data type. If implemented for user defined type, then the code has to be as follows:

```
Number s;
cin>> s;
```

s is an object of class **Number** and cin is an object of **istream**. In this statement left hand side operand is not an object of user defined class Number. So the operator >> cannot be overloaded using the member function of class Number. It should be implemented as friend function.

3.9 RULES FOR OPERATOR OVERLOADING

- No new operators can be created, only existing operators can be overloaded.
- The overloaded operator must have at least one operand that is of user defined type.
- Precedence and Associativity of an operator cannot be changed.

- Change the basic meaning of an operator is not possible. For example, redefining a plus(+) operator to subtract one value from the other is not possible
- Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
- "friend" functions cannot be used to overload certain operators like
    - assignment operator „ = "
    - function call operator „( )"
    - subscripting operator „[ ]"
    - class member access operator „->"
- Member function can be used to overload the above operators.
- The following operators cannot be overloaded:
    - scope operator - **::**
    - member selector - **.**
    - member pointer selector - *
    - ternary operator - **?:**
- Numbers of Operands cannot be changed. That is, Unary operator remains unary, binary remains binary etc.
- Unary operators, overloaded by a member function, take no explicit arguments and return no explicit values.
- Unary operators, overloaded by a friend function, take one reference argument - the object of the relevant class.
- Binary operators overloaded through a member function take one explicit argument and through a friend function it takes two explicit arguments.
- Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.

3.10 TYPE CONVERSION

Implicit conversions are automatically performed when a value is copied to a compatible type.

standard conversion - Standard conversions affect fundamental data types, and allow the conversions between numerical types

- short to int,
- int to float
- double to int
- to or from bool etc.,

For example:

```
short a=100;
int b;
b=a;
```

In the above example, the value of „aa is promoted from short to int without the need of any explicit operator.

promotion - Converting from a data type with small size to another larger data type guarantees the exact same value in the destination type. Example:

- short to int
- float to double

- Converting from a data type with large size to another smaller data type does not guarantee the exact same value in the destination type. Example:
  - float to int : the value is truncated by removing the decimal part.
- If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation
- The conversions from/to boolean consider false equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.
- For non-fundamental type conversions like arrays and functions it implicitly converts to pointers. Pointers normally allow the following conversions:
  - *Null pointers* : converted to pointers of any type
  - Pointers to any type: converted to void pointers.
  - Pointer *upcast*: pointers to a derived class: converted to a pointer of an *accessible* and *unambiguous* base class.

Implicit conversions with classes

Implicit conversions can be controlled by the following member functions:

- ***Single-argument constructors:*** allow implicit conversion from a particular type to initialize an object.
- ***Assignment operator:*** allow implicit conversion from a particular type on assignments.

- ***Type-cast operator:*** allow implicit conversion to a particular type.

Three types of situations might arise for data conversion between different types :

(i) Conversion from basic type to class type.

(ii) Conversion from class type to basic type.

(iii) Conversion from one class type to another class type.

(i) Basic Type to Class Type

This type of conversion is very easy. If a class object has been used as the left hand operand of = operator, the type conversion can also be done by using an overloaded = operator in C++.

(ii) Class Type to Basic Type

Define an overloaded casting operator for converting a class type to a basic type. The syntax of the conversion function is as follows:

```
operator typename()
{
.......
. ......//statements
}
```

- The function converts a class type data to typename. For example
    - The operator float ( ) converts a class type to type float
    - The operator int ( ) converts a class type object to type int.
- When a class type to a basic type conversion is required, the compiler will call the casting operator function for performing the task.
- The following conditions should be satisfied by the casting operator function :
    - It must not have any argument
    - It must be a class member
    - It must not specify a return type.

(i) One Class Type to Another Class Type

- During converting one class type data A to another class type data B, A is referred to as the Source class and b as Destination class.
- The source class performs the conversion and result is given to the object of destination class.

- The argument of the source class is passed to the destination class for the purpose of conversion.
- The conversion constructor must be kept in the destination class.
- The conversion can be performed in two ways :
    (a) Using a constructor.
    (b) Using a conversion function.

* * *

UNIT IV

4.1 INHERITANCE: DEFINING DERIVED CLASSES

Inheritance is the process by which objects of one class acquires the properties of objects of another classes. It supports the concept of hierarchical classification. It has the capability of one class to acquire properties and characteristics from another class.

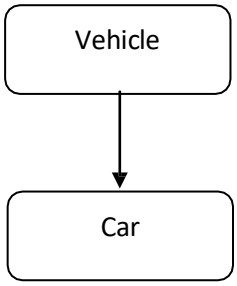
- The class whose properties are inherited by other class is called the ***Parent*** *or* ***Base*** *or* ***Super class.***
- The class which inherits properties of other class is called ***Child*** *or* ***Derived*** *or* ***Sub class***.
- When an existing class is inherited, all its methods and fields become available in the new class, making the code reusable.
- All members of a class except Private can be inherited.
- Additional features can be added to an existing class without modifying it making it possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.

Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

- Before defining a subclass, the super class must be defined or must be declared before the subclass declaration.
- The public, privtate or protected Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass.

Example of Inheritance



```
class Vehicle
{ public:
  int wheels = 4;
};

class Car : public Vehicle
{ public:
  int speed = 100;
};

int main()
{
 Car c;
 cout << c.wheels;
 cout << c.speed;
}

Output :
4 100
```

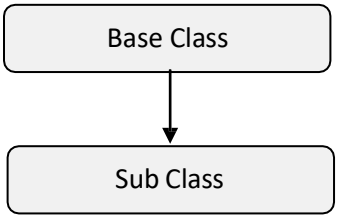
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)

4.2 SINGLE INHERITANCE

One derived class inherits from only one base class. It is the most simplest form of Inheritance.



Single Inheritance can inherit properties by the following three access methods:

- Private Inheritance
- Public Inheritance
- Protected Inheritance

Private Inheritance

Consider the following classes:

```
class A
{
.....
…
….
};

class B: private A
{
…
…
…
}
```

- In the above class definition, all the public parts and all the protected parts of class A become private members/parts of the derived class in B.
- No private member of class A can be accessed by class B.
- A public function can be accessed by any object, but private function can be used only within the class hierarchy.

Public Inheritance

Consider the following classes:

```
class A
{
.....
……
};

class B: public A
{
…
…
};
```

In the above class definition, all the public and protected parts of class A become public and protected class in class B respectively.

Protected Inheritance

Consider the following classes:

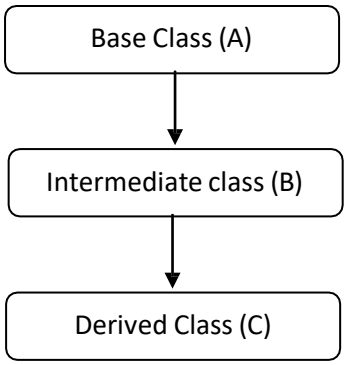
```
class A
{
.....
……
};

class B: protected A
{
 …
…
};
```

Now, all the public and protected parts of class A become protected in class E.

No private member of class A can be accessed by class E.

4.3 MULTILEVEL INHERITANCE

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



From the base Class A, Class B is derived. Class B serves as the Base class for Class C. In this case, Class B is called as the Intermediate Base Class which provides a link for the inheritance between Class A and Class C

Multilevel Inheritance declaration:

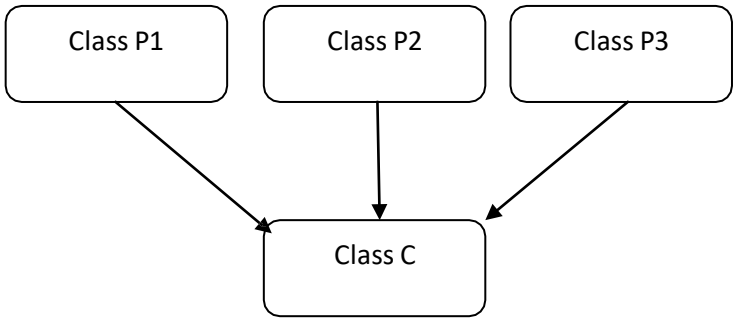
```
class A
{ .....
…       };

class B: public A
{ …
…       };
```

```
class C: public B
{ …
…      };
```

The chain A→ B → C is called as the inheritance path. This process of creating derived class can be extended to any number of levels. In multilevel inheritance the constructors are executed in the order of inheritance.

4.4 MULTIPLE INHERITANCE

In this type, a Single Class can inherit the attributes from many Classes. It allows the user to combine the features of existing classes from which it was inherited. The derived class may contain the features of all the parent classes.



The class declaration is as follows:

```
class C: access_mode P1, access_mode P2…
{ .....
…      };

class P1
{ …
…      };

class P2
{ …
…      };
```

The ***access_mode*** can be either public or private. The parent classes are separated by comma.

In Multiple inheritance, the base classes are constructed in the order its declaration in the derived class.

Advantages of Inheritance

- Code Reusability
- Method Overriding

- Use of Virtual Keyword

4.5 POINTERS : DECLARING AND INITIALIZING POINTERS

Pointers are used in C++ for efficient memory management and for applying the concept of polymorphism.

C++ Pointer Declaration

The general form of a pointer declaration is as follows :

```
type *var_name ;
```

where type is any valid C++ data type and var_name is the name of the pointer variable. Following declarations declares pointers of different types :

*int *ptr ;*	creates an integer pointer pointing to an integer.
*char *ptr ;*	creates a character pointer pointing to a character
*float *ptr ;*	creates a float pointer pointing to a float data type.

- Two special operators * and & are used with pointers.
- The & is a unary operator that returns the memory address of its operand.

Example,

```
int p = 100 ;
int *ptr ;
ptr = &p;
```

In the above example, the memory address of p is stored into ptr, i.e ptr points to the memory location of p. The expression &p will return an int pointer value because p is an integer.

- Using '*' operator, changing/accessing value being pointed to by pointer is called Dereferencing. Thus Dereferencing refers to changing/accessing state of the pointer.
- The pointer operators „&“ and „*“ have a higher precedence than all other arithmetic operators except the unary minus, which has equal precedence.

C++ Pointers Initialization

- A pointer variable must not remain un-initialized. It must be initiliazed atleast with a NULL pointer value.
- The zero pointer NULL is defined in the standard file stddef.h.
- Uninitialized pointers cause the system crash.

- nullptr keyword in C++ is a null pointer constant that can be used to initialize pointers. Example:

```
char *chptr = nullptr ;
int *inptr = nullptr ;
```

Difference between NULL and nullptr

Difference between NULL and nullptr are:

NULL	Nullptr
Not defined by C++	Defined by C++ as a legal null pointer
Internally mapped to int 0(zero)	It is a legal empty/null pointer

4.6 POINTERS EXPRESSION AND POINTER ARITHMETIC

Base Address - A pointer holds the address of the very first byte of the memory location where it is pointing to. The address of the first byte is known as Base Address.

Pointer expressions

- Incrementing using ***++ptr* or *ptr++***
- Decrementing using ***--ptr* or *ptr—***
- Example:

```
int n[10] ;
int *nptr ;
nptr=&n[0];
```

In the above expression, the pointer variable ***nptr*** refers to the base address of the variable n. The expression ***nptr++*** moves the pointer to the next memory address. If decremented using ***nptr—***it moves the pointer to the previous memory address.

- When a character pointer is incremented, its value increases by 1 (if the size of char is one on the machine).
- An integer pointer when incremented, its value increases by two (or the size of int on the machine);

- a float pointer when incremented, its value increases by eight (or the size of float on the machine).

Pointer Arithmetic

- Only two arithmetic operation, addition and subtraction, may be performed on pointers.
- A pointer either incremented or decremented for addition or subtraction respectively.
- Any integer can be added or subtracted from a pointer.
- Each time a pointer is incremented by 1, it points to the memory location of the next element of its base data type.
- Example: If ***inptr*** an integer pointer is currently pointing to memory address 5001 and the int size is 2 bytes, then after the following expression ***inptr++; inptr*** will be pointing to 5003 and not to 5002. Each time you increment iptr by 1, it will point to the next integer. Similarly, when you decrement iptr by 1 it will point to the location of the previous element of its base type (int here) which is at address 999.
- Pointer arithmetic on variables that are not stored in contiguous memory location is not possible.

4.7 USING POINTERS WITH ARRAYS AND STRINGS

- Pointers can be used to access the elements of an array.
- Pointers can be used to allocate arrays dynamically.
- It decides the array size at runtime.
- ***malloc( )*** and ***calloc( )*** functions are used for memory allocation.
- Difference between array and pointer

Arrays	***Pointers***
Refers to a block of memory space.	Do not refer to any section of memory.
Memory addresses of array cannot be changed	The memory address that a pointer refers to can be changed.

- An array of pointers that represent a collection of addresses can be created to save memory space.

- Example for declaring array of pointers, each of which points to an integer:

```
int *narray[20] ;
```

This declaration causes an allocation of contiguous memory for 10 pointers that can point to integers. To each of the pointers, the elements of pointer array, may be initialized.

C++ Pointers and Strings

- A string is a one-dimensional array of characters terminated by a null ('\0').
- Strings and their constituent characters can be accesses using char pointers.
- An array of char pointers is normally for storing several strings in memory.
- Example : declaring many strings using array of pointers;

```
char *names[] = {"AAA", "BBB", "CCC", "Y"} ;
```

In the above declaration names[] is an array of char pointers whose element pointers contains base address of respective names. That is, the element pointer names[0] stores the base address of string "AAA", the element pointer names[1] stores the above address of string "BBB", and so forth.

- The array of char pointers is generally preferred over two dimensional array of characters because of the following two :
    - An array of pointers makes more efficient use of available memory by consuming lesser number of bytes to store the strings.
    - Exchange the positions of strings in the array using pointers can be done easily.

4.8 *„this"* POINTER

- ***this*** is a unique keyword to represent an object that invokes a member function.
- ***this*** pointer is a hidden pointer inside every class member function that points to the class object.
- ***this*** pointer is automatically passed to a member function when it is called.
- ***this*** pointer acts as an implicit argument to all the member functions.

- ***this*** pointer is a constant pointer that holds the memory address of the current object.
- ***this*** pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.
- ***this*** pointer is not available in static member functions as static member functions can be called without any object (with class name).
- Example:

```
class XYZ
{
int a;
…..
….. };

{ …
this->a = 100;
….
return *this;
…
}
```

- ***this*** pointer is used under the following situations:
    - When local variable‟s name is same as member‟s name
    - To return reference to the calling object

4.9 POINTERS TO DERIVED CLASSES

- Pointers can be used to the objects of derived classes.
- Type-compatible: Pointers to objects of base class are type-compatible to the objects of a derived class.
- A single pointer variable can point to objects of different classes.
- Using pointer only the data members inherited from the base class can be accessed. Members that belong only to derived class cannot be accessed.

- Example:

```
X *ptr;  // pointer to class X
X x;     // base object
Y y;     // derived object
ptr =&x; // pointer points to base object
ptr = &d; // pointer points to derived object
```

- A base pointer can point to any number of derived objects, but it cannot directly access the members defined by the derived class.

* * *

INSTRUCTOR: Engr. NFORBINEH MABEL

UNIT V

5.1 VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS

Virtual Functions

In object-oriented programming, a virtual function is a function or method whose behavior can be overridden within an inheriting class by a function with the same name. This concept is carried out using polymorphism.

Polymorphism mechanism is supported in C++ by the use of virtual functions. The concept of virtual function is related to the concept of dynamic binding. Binding refers to binding of actual code to a function call. Dynamic binding is a binding mechanism in which the actual function call is bound at run-time and it is dependent on the contents of function pointer at run time. By altering the content of function pointers, different functions having a same name but different code can be called, which exhibits polymorphic behaviour.

Polymorphism means „one name multiple forms". Runtime polymorphism can be achieved by using virtual functions. The virtual function implementation in C++ is shown as in Figure 5.1.

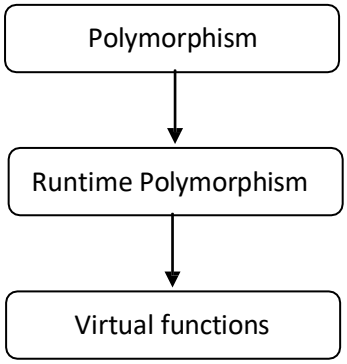


Fig. 5.1 Virtual functions from Polymorphism

- When the same function name is used in both the base class and the derived class, the function in base class is declared as virtual using virtual keyword preceding its normal declaration.
- The function to be used at runtime is determined based on the type of object pointed to by the base pointer.
- If the base pointer points to different objects, different versions of the virtual fuction can be executed.

Example:

```
class base
{ …
virtual void func( )
  { ….. }
… };

class derived : public base
{
…
void func( )
    { …. }
};

int main( )
{
base B;
derived D;
base *bptr;
bptr = &B;
…
bptr ->func( );// calls base version
…
bptr = &D;
…
bptr ->func( );// calls derived version
…
}
```

- Virtual function can be accessed through the use of a pointer declared as a pointer to the base class.
- Runtime polymorphism is achieved only when a virtual function is accessed through the use of a pointer to the base class.

Rules for virtual functions:

- Virtual function in a base class must be defined whether used or not.
- Accessed using object pointers.
- Must be a member of a class.
- Virtual functions cannot be static members.
- Can be a function of another class.
- Cannot have virtual constructors
- Can have virtual destructors

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with= 0. The syntax for a pure virtual function is

```
virtual void func() = 0;
```

A pure virtual function is a function that has the notation "= 0" in the declaration of that function.

```
class class_name
{ …
        public:
   virtual void pure_virtual( ) = 0;  // no function body
…
 };
```

- The **"= 0"** of a pure virtual function is called as the ***pure specifier.***
- The notation "= 0" indicates that the virtual function is a pure virtual function, and has no body or definition.

```
// implementation of pure virtual function
{ …
        public:
   virtual void pure_virtual( ) = 0;  // no function body
…
 };
void class_name::pure_virtual( )
{
    …..;
}
```

- Pure virtual functions cannot have a definition inside the function declaration
- A class with a pure virtual function is called an abstract class.

- Pure Virtual functions cannot create object of Abstract class.
- Pure Virtual function must be defined outside the class definition. If defined inside the class definition, complier error occurs.
- A pure virtual function declared in a base class has no definition relative to the base class.
- A class containing a pure virtual function cannot be used to declare any object of its own.

5.2 WORKING WITH FILES

The data is stored in devices like hard disk using the concept of files. A file is a collection of related data stored in a particular area on a disk. Programs can be designed to perform the read and write operations on these files. (Figure 5.2)

I/O Stream: The I/O system of C++ handles file operations. It uses file streams as an interface between the programs and files. The stream that supplies data to the program is called input stream and the one that receives data from the program is called output stream.

Input stream:

- Input stream extracts data from the file.
- The input operation involves the creation of an input stream and linking it with the program and input file.

Output stream:

- Output stream inserts data to the file.
- The output operation involves establishing an output stream with the necessary links with the program and output file.

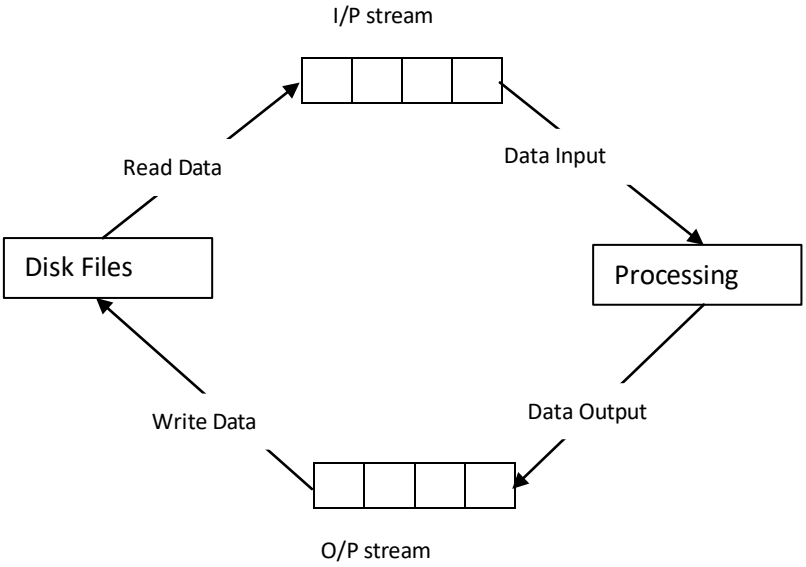


Fig. 5.2 I/O stream & file operations

5.3 CLASSES FOR FILE STREAM OPERATORS

The I/O system of C++ contains a set of classes that defines the file handling methods. These include ***ifstream, ofstream and fstream***. These classes are derived from ***fstreambase*** and from the corresponding ***iostream*** class. These classes are declared in ***fstream*** and therefore

this file is included in any program that uses files. The various stream classes for console I/O operations are shown in Figure 5.3.

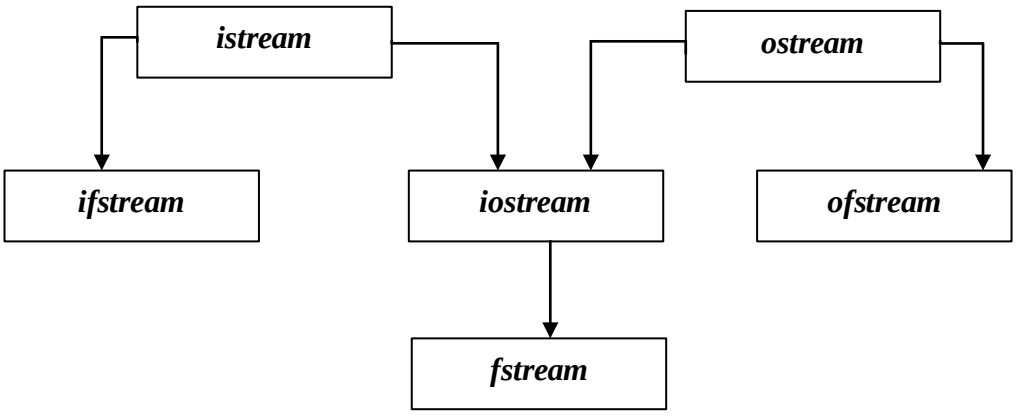


Fig 5.3 file stream classes

The details of file stream classes are described in Table 5.1.

Table 5.1 File stream classes and its functions

ios	- Provides basic facilities for I/O classes - Contains pointer to a buffer object:***streambuf*** - Declares constants and functions for handling I/O operations.
istream	- Declares input functions like ***get( ), read( ).*** - Contains overloaded extraction operator >> - Inherits the properties of ***ios***
ostream	- Declares input functions like ***put( ), write( ).*** - Contains overloaded extraction operator >> - Inherits the properties of ***ios***
ifstream	- Provides input operations. - Contains **open()** with default input mode. - Inherits the functions ***get(),getline(),read(),seekg(),tellg()*** functions from ***istream***.
ofstream	- Provides output operations. - Contains ***open()*** with default output mode. - Inherits ***put(),seekp(),tellp()*** and ***write()*** functions from ***ostream.***
fstream	- Provides support for simultaneous input and output operations. - Contains open with default input mode. - Inherits all the functions from **istream** and **ostream** classes through **iostream**.
Streambuf	- Provides an interface to devices through buffers - Acts as a base for ***filebuf*** class
iostream	- Inherits the properties of ***ios, istream*** and ***ostream*** through multiple inheritance - Contains all I/O functions

5.4 OPENING AND CLOSING A FILE

For using a disk file the following things are necessary:

- Suitable file name
- Data type and structure
- Purpose
- Opening Method

The filename is a string of characters that makeup a valid filename for the operating system. It contains two parts

- primary name
- period (optional) with extension.

Examples: abc.txt, myfile.doc

For opening a file initially a file stream is created and then it is linked to the filename. A file stream can be defined using the classes ***ifstream, ofstream*** and ***fstream*** that are in the ***fstream*** header file. Table 5.2 describe the contents of file classes.

Table 5.2 contents of file stream classes.

filebuf	- Purpose is to set the file buffers to read and write. - Has ***close()*** and ***open()*** as members.
fstreambase	- Provides operations common to file streams. - Serves as a base for ***fstream, ifstream*** and ***ofstream*** class. - Contains ***open()*** and ***close()*** functions.

The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file.

A file can be opened in two ways:

- using the constructor function of the class
- using the member function open( ) of the class

Opening Files using Constructor:

While using constructor for opening files, filename is used to initialize the file stream object. This involves the following steps:

- Create a file stream object to manage the stream using the appropriate class.
  - class ***ofstream*** for output stream creation
  - class ***ifstream*** for input stream creation.

- Initialize the file object using desired file name.v

Example:

```
ifstream infile("inmyfile"); //input only
```

opens a file named "***inmyfile***" for input. This creates infile as an **ifstream** object that manages the input stream for reading data. Similarly,

```
ofstream outfile("outmyfile"); //output only
```

opens a file named "***outmyfile***" for output. This create outfile as an **ofstream** object that manages the output stream. The figure 5.3 illustrates the file stream working on two separate files.

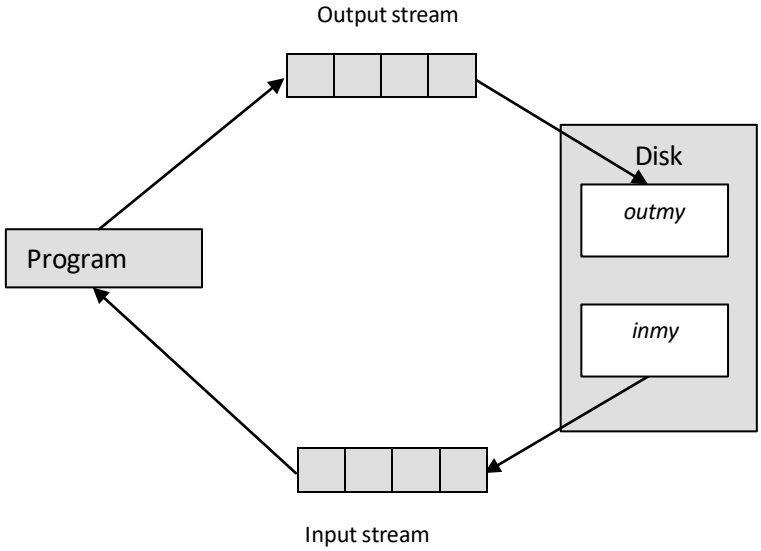


Fig 5.4 I/O File streams on separate files

Example: Creating files with constructor function

```
#include <iostream.h>
#include <fstream.h>
int main()
{
ofstream outfile("RESULT");
…
cin >>name;
outfile <<name <<"\n";
…
cin >>mark;
outfile <<mark <<"\n";
outfile.close();
```

```
ifstream infile("mark");
infile >>name;
infile >>mark;
…
infile.close();
return 0;
}
```

Opening Files using open()

- **open()** function can be used to open multiple files that uses the same stream object.
- For processing many files sequentially, a single stream object can be created and can be used to open each file in turn.
- General syntax:

```
File-stream-class-name stream-object-name;
stream-object-name.open ("filename");
```

Example: for working with multiple files simultaneously:

```
//Creating files with open() function
#include <iostream.h>
#include<fstream.h>
     int main()
    {
     ofstream fileout;
     fileout.open("file1");
     fileout<<"…\n";
   ….
     fileout.close();
fileout.open("file2");
fileout<<"….\n";
...
fileout.close();

…
ifstream filein;
                                    filein.open("file1");
cout<<"contents of file1 \n";
while (filein)
{
                                filein.getline(arg1,arg2);
cout<<arg1;
}
```

```
                                filein.close();
                                filein.open("file2");
cout<<"contents of file2";
while(filein)
{
                        filein.getline(arg1,arg2);
cout<<arg2;
}
                    filein.close();
return 0;
}
```

The Figure 5.5 illustrates the file streams working on multiple files concept. At a particular time a stream can work on one file only. To work on the current file all the other files has to be closed.

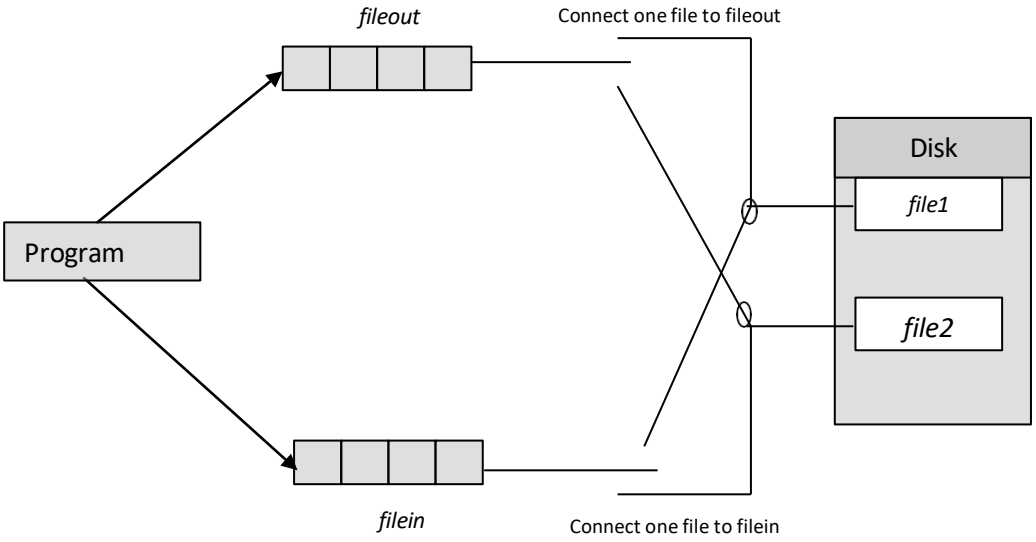


Fig 5.5 File streams working on multiple files

5.5 CHECKING FOR eof( )

While reading a data from a file, it is necessary to find where the file ends i.e end of file. If the end of file is not detected, the program drops in an infinite loop. To avoid this, it is necessary to provide appropriate instructions to the program for detecting end of file condition.

- When end of file is detected, the process of reading data from that file can be terminated.
- An ***ifstream*** object returns a value of 0 during the end-of –file condition.
- The following methods can be used to detect end-of-file condition:

- A ***while*** loop can be made to terminate when the file stream object returns a value of zero on reaching the end-of –file condition.
- Using ***if*** condition the condition can be checked as follows:

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

It returns a non zero value if end of file condition is encountered and zero otherwise .Therefore the above statement terminates the program on reaching the end of file.

5.6 FILE MODES- GET( ) AND PUT( ) FUNCTIONS

The general form of opening a file can be described as follows:

```
stream_object.open("filename",file_mode);
```

The file_mode parameter specifies the purpose for which the file is opened. The three important file modes are

- reading
- writing
- appending

The default values are

ios::in - reading only

ios::out writing only

The file mode parameter can take constants defined in class **ios**. The following table lists the file mode parameter and their meanings.

Table 5.2 File Mode Operation Parameter

File Mode Parameter	***Purpose***
ios::app	*Appending to end-of-file*
ios::ate	*Going to end-of-file on opening*
ios::binary	*Binary file*
ios::in	*Opening file for read only*
ios::nocreate	*Opening fails if file does not exist*
ios::out	*Opening file for write only*
ios::trunk	*Delete the contents of the file if it exists*

5.6 ***get( ) AND put( )FUNCTIONS***

The file stream classes has member functions for performing the input and output operations on files. The functions ***put()*** and ***get()*** are designed for handling a single character at a time.

- ***put()*** **-** writes a single character to the associated stream.
- ***get()*** **-** reads a single character from the associated stream.

Example:

```
int main()
{
…
char str[80];
fstream file;
file.open("myfile", ios::in | ios::out); // open file
…
…
file.put(str[i]);
…
char c;
…
file.get(c);
cout<<c;
…
return 0;
}
```

5.7 BINARY FILES – write( ) and read( ) functions

The functions ***write()*** and ***read(),*** handle the data in binary form. The values are stored in the internal memory in the binary form. If the data is to be stored inside the disk in the binary form read( ) and write( ) functions can be used. If we want to store an integer in binary format it needs two bytes whereas it needs four bytes to store it in character format. (Figure 5.6 & fig 5.7)

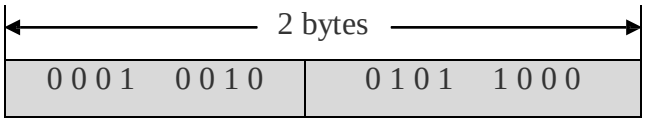


Fig 5.6 Binary form

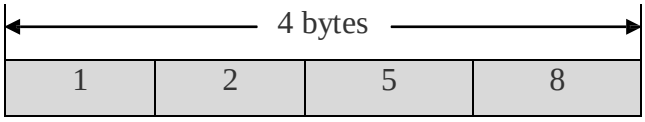


Fig 5.7 Character format

Irrespective of its size, an integer (int) character takes two bytes to store its value in the binary form. At the same time, a 4 digit (int) will take four bytes to store it in the character form. The binary input and output functions take the following form:

```
file1.read (( char * ) & V, sizeof (V));
file2.write (( char *) & V, sizeof (V));
```

- The functions take two arguments - the address of the variable V, and the length of that variable in bytes.
- The address of the variable is pointer to character type. It must be cast to type char*

Example:

```
int main( )
{
…
float var;
file1.write (( char *) & var, sizeof (var));
…
file2.read (( char * ) & var,  sizeof (var));
…
}
```

5.8 ERROR HANDLING

There are many problems encountered while dealing with files. Some of the commonly occurring problems are as follows:

• Attempting to open a a file that does not exist.

• Attempting to use the same file name that already exist.

• Attempting an invalid operation on file.

• Attempting to store the file where there is no disk space.

• Attempting to use an invalid file name.

• Attempting to perform an operation that is invalid under that file mode.

ios class:

- The class ***ios*** has member functions that can be used to read the status recorded in a file stream.
- The file stream inherits a ***„stream-state"*** member from the class ***ios*** which records information on the status of the currently used file.

- The stream state member uses bit fields to store the status of error conditions.

Table 6.4 Error Handling Functions

Function	**Return value and meaning**
eof()	If end of file is encountered during reading then return TRUE else return FALSE.
fail()	If an input or output operation has failed then return TRUE.
bad()	Returns TRUE under the following conditions: - an invalid operation is attempted - any unrecoverable error has occurred.
good()	Returns TRUE if error has not occurred and the system can proceed to perform I/O operations.

Example:

```
{
…
ifstream myfile;
myfile.open("XYZ");

…
if myfile.eof()   // checking end of file condition}
..
if myfile.bad() // checking for invalid operations
…
while (myfile.read(…) ) // checking for invalid operations
```

* * *