Unit II - CLASSES AND OBJECTS

A class is an abstract idea that can be represented with data structures and functions. Functions associated with a class are called methods. An object is said to be as instance of a class.

A class combines data and functions (procedures) into a single programming unit. This is ‘encapsulation’. Data and procedures are tied together logically. The parts that make up a class can be protected from the rest of the program to varying degrees. The programmer has control over the protection of individual entries in the class definition.

Private and Public

A key feature of object-oriented programming is ‘data-hiding’ ie., the data is concealed within a class, so that it cannot be accessed mistakenly by functions outside the class. ‘private’ and ‘public’ are the two types of protection available within a class.

Items marked with **private** can only be accessed by methods defined as part of the class. Data is most often defined as private. Private members can be accessed by members of the class. **Public** items can be accessed from anywhere in the program without restrictions. Class methods are usually public. As a general rule, data should not be declared public. Public members can be accessed by members and objects of the class.

Protected

This is another type of protection available within a class. Items declared as protected are **private** within a class and are available for private access in the derived class.

Structures and Classes

The only formal difference between a class and structure is that in a class the members are private by default, while in a structure they are public by default. Structures can include functions but the commonly practiced way is to use the structures for data structures and classes for member functions.

Specifying A Class

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

1. Class declaration
2 .Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

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

The **class** declaration is similar to a **struct** declaration. The keyword class specifies, that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called class members. They are usually grouped under two sections, namely, **private** and **public** to denote which of the members are private and which of them are public. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword **private** is optional. By default, the members of a class are private. If both the labels are missing, then by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as data members and the functions are known as member function. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. The binding of data, and functions together into a single class-type variable is referred to as encapsulation.

A Simple Class Example

A typical class declaration would look like:

```
class Item

{
int number;// variables declaration
float cost;// private by default
public:
void getdata(int a, float b);// functions declaration
void putdata(void);// using prototype
}:// ends with semicolon
```

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare instances of that class type. The class item contains two data members and two function members. The data members are private by default while both, the functions are public by declaration. The function getdata() can be used to assign values to the member variables number and cost, and putdata() for displaying their values. The data members are usually declared as private and the member functions as public.

Creating Objects

Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

Item p; - creates a variable p of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item. We may also declare more than one object in one statement. Example

```
Item x,y,z;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition,

```
class Item
{
    .....
.....
    .....
} x,y,z;
```

would create the objects x, y and z of type **Item** . This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

Private data of a class can be accessed only though the member functions of that class. The main() cannot contain statements that access number and cost directly. The following is the format for calling a member function:

object-name.function- name (actual -arguments);

For example, the function call statement, **x.getdata(100,75.5);**is valid and assigns the value 100 to number and 75.5 to cost through a and b function parameters of the object x by implementing the getdata() function.

Similarly, the statement **x.putdata();**would display the values of data members. Remember, a member function can be invoked only by using an object of the same class.

Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. An important difference between a member Function and a normal function is that a member function incorporates a membership 'identity label' in the header. This label tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return- type class-name :: function-name (argument declaration)
{
   Function body
}
```

The membership label class-name :: tells the compiler that the function *function- name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name*s pecified in the header line. The symbol :: is called the scope resolution operator.

For instance, consider the member functions **getdata()** and **putdata()**, they may be coded as follows:

```
void Item :: getdata(int a, float b)
{number = a;
cost=b;}

void Item :: putdata(void)
{ cout<< "Number e" *c number << "\n";
  cout<< "Cost :"<<cast<< "\n":}
```

Since those functions do not return any value, their return-type is void. The member functions have some special characteristics that are often used in the program development. These characteristics are:

- Several different classes can use the same function name. The membership label' will resolve their scope.
- Member functions can access the private data of the class. A non- member function cannot do so normally.
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the Item class as follows:

```
class item
{
int number;
float cost;
public:
void getdata(int a, float b); // declaration
// inline function
void putdata(void) // definition inside the class
{
cout << numher << "\n";
cout <<cost<<"\n";
}
};
```

When a function is defined inside class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an inline Junction are also applicable here. Normally, only small functions are defined inside the class definition.

Nesting of Member Functions

Normally in object oriented programs, member function of a class can be called only by an abject of that class using a dot operator. However, there is an exception to this a member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions. Example,

```
void Item :: getdata(int a, float b)
{number = a;
cost=b; putdate();//Nesting of member function}

void Item :: putdata(void)
{ cout<< "Number e" *c number << "\n";
  cout<< "Cost :"<<cast<< "\n":}
```

Private Member Functions

Although it is normal practice to place all the data items in a private tertian and all the functions in public, some situations require certain functions to be hidden (like private date) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

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

Arrays within a Class

The arrays can be used as member variables in a class. Using member functions data can store in these variables and process it. For example students marks can store in an array using single variable with subscript value. The marks variable looks like, int marks[6];

Memory Allocation for Objects

The memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created .Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

Static Data Members

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

Static Member Functions

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

- It is visible only within the class, but its lifetime is the entire program.

Static variables arc normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

Static Member Functions

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

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the-class name (Instead of its objects) as follows:
-

class-name :: function-name();

Arrays of objects

An array can be of any data type including **struct**. Similarly, it can also have arrays of variables that are of the type **class**. Such variables are called arrays of objects. The complete data and functions are fully assigned to each array object while generating object arrays. Here the object is used with an index or subscript to refer the actual object in the array.

Using single object specification, number of objects can represent in arrays. All are homogeneous type of objects, because array objects are generated from a single class type. Such a number of arrays can generate independently from different class types. All these object arrays are homogeneous types.

Objects as Function Arguments

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

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

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Friendly Functions

One thing is emphasizing that the private members cannot be accessed from outside the class. That is, a non -member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes have been defined. We would like to use a function to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

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

```
Class class_name
{
        ........
        ........
public:
        ........
        ........
friend function_return_type function_name (Parameters);  // declaration
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword friend or the scope operator. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name. (eg. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

This method can be used to alter the values of the private members of a class. Altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Unit III – Constructors and Destructors, Overloading

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. It also provides another member function called the destructor that destroys the objects when they are no longer required.

Constructors

Automatic initialization is carried out using a special member function called the **constructor**. It is a convenient way to initialize an object when it is first created without the need to make a separate call to a member function. Thus, a constructor is a member function that is executed automatically whenever an object is created. Constructors may or may not take arguments, depending on how the object is to be constructed. There is no return value from the constructors.

A constructor always has the same name as the class itself. Every class has an implicit constructor which need not be defined. It is called automatically for the object as it is created. Constructor functions can be overloaded.

A constructor is declared and defined as follows:

```
class Integer
int m, n:
public:
        Integer(void); // constructor declared
        ............
        ............
};
Integer : : Integer(void) // constructor defined
{m=0; n= 0;  }
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration,

```
Integer intl;   // object int1 created
```

not only creates the abject int1 of type **Integer** but also initializes its data members **m** and ***n*** to zero There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the default constructor. The default constructor for class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor, Therefore a statement such as,

A a; invokes the default constructor of the compiler to create the object *a*.

The constructor functions have some special characteristics. These are:

• They should be declared in the public section.

• They are invoked automatically when the objects are created.

• They do not have return types, not even void and therefore, and they cannot return values.

• They cannot be inherited, though a derived class can call the base class constructor.

• Like other C++ functions, they can have default arguments.

• Constructors cannot be virtual.

• They make implicit calls' to the operators new and delete when memory allocation is required.

Parameterised Constructor

The constructor Integer(), defined above, initializes the data members of all the objects to zero, However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

The constructor Integer() may be modified to take arguments as shown below:

```
class Integer
{
                int m,n;
public:
        Integer (int x, int y) // parameterized constructor
        ……...
        ………
};
Integer : : Integer (int x, int y)
{
        m = x; n = y;
}
```

When a constructor has been parameterized, the abject declaration statement such as

```
        Integer intl;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
        Integer intl = Integer (0,100) ; // explicit call
```

This statement creates an integer object intl and passes the values 0 and 100 to it. The second implemented as follows:

```
        Integer intl(0,100); // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement. Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor.

Multiple Constructors in a Class

In a class it may contains more than one constructors that is default and number of parameterised constructors. But, each constructor is related to its corresponding object with its parameters. In the case of default constructor, there is no argument there while generating object. In the case of parameterised constructor, that may contains one or more parameters as basic type values or class objects. This is the method of creating multiple constructors.

For example,

```
class Integer
{
        int m, n;
public:
        Integer(){m=0; n=0,}  //constructor 1
        Integer(int a, int b
        {m=a; n=b;}           //constructor 2
        Integer (Integer &i)
        { m=i.m; n=i.n; }     //constructor 3
```

This declares three constructors for an Integer object. The first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as an argument. For example, the declaration

```
        Integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
        Integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to 20 and 40 respectively, Finally, the statement

```
        Integer I3 (I2);
```

would invoke the third constructor which copies the values of I2 into I3. In other words, it sets the value of every data element of I3 to the value of the corresponding data element of I2. When more than one constructor function is defined in a class, we say that the constructor is overloaded.

Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor Complex() can be declared as follows:

    Complex (float real, float imag=0);

The default value of the argument **Imag** is zero. Then, the statement

    Complex C(5.0);

assigns the value 5.0 to the **real** variable and 0.0 to **imag**(by default). However, the statement

    Complex C(2.0,3.0);

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. The missing arguments must be the trailing ones.

Copy Constructor

Copy constructor is used to declare and initialize an object from another object. The general form of this method is specified in a class is given below. Here Integer is a class.

    Integer(Integer &i);

The following statement defines the object I2 and at the same time initializes it to the values of I1.

    Integer I2(I1);

Another form of this statement is, **Integer I2 = I1**;

The process of initializing through a copy constructor is known as copy initialization. Remember, the statement, **I2 = I1**will not invoke the copy constructor. However, if I1 and I2 are objects, this statement is legal and simply assigns the values of I1 to 12, member-by-member. This is the task of the overloaded assignment operator(=).

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

Dynamic Constructor

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the **new** operator. Another operator **delete** is used for the deactivation of used memory after its use. For example, **Name** is string variable and its length is assigned to the numeric variable **Length**. Then

```
Name = new char[Length+1];
………
………
DeleteName;
```

DESTRUCTORS

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~). For example, the destructor for the class Integer can be defined as shown below:

```
~Integer() { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case maybe) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use **delete** to free that memory.

Operator Overloading

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. It tries to make the user-defined data types to have in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with, the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (.*).
- Scope resolution operator (::).
- Size operator (sizeof).
- Conditional operator (?:).

The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
return_type class_name ::operator op( argument_list)

{

        Function body // task defined

}
```

where *return typ*e is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword *operator*.operator *op* is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
Vector operator + (Vector);                    // vector addition

Vector operator - (Vector);                    // vector minus

friend Vector operator + (Vector, Vector);     // vector addition

friend Vector operator - (Vector);             // unary minus

Vector operator - (Vector &a);                 // subtraction

int  operator == (Vector);                     // comparison

friend int operator == (Vector, Vector);       // comparison
```

Vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function **operatorop**() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

op x or *x op*

for unary operators and

x op y

for binary operators. op x (or x op) would be interpreted as

operator op(x)

for **friend** functions. Similarly ,the expression *x op y* would be interpreted as either

x.operator op (y)

in case of member functions, or

operator op (x,y)

in case of friend functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say we cannot redefine the plus (+) operator to subtract one value from the other.
4. Overloaded Operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as+,-,*, and / must explicitly return a value. They must not attempt to change their own arguments.

Type Conversions

When constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert x to an integer before its value is assigned to m. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

In user-defined type the conversion operation is conducted in another way. Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = vl + v2; // vl, v2 and v3 ore class type objects.
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. In the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. If one of the operands is an object and the other is a built-in type variable, this will not work as a smooth operation.

Since the user-defined data types are defined to suit the requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between incompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

Basic to Class Type

The conversion from basic typo to class type is easy to accomplish. It may be recalled that the use of constructors to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, another constructor is used to build a string type object from a char* type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type. Consider the following constructor:

```
string ::string(char *a)
        {
                length = strlen(a);
                P = new char[length+l];
                strcpy(P,a);
        }
```

This constructor builds a **string** type object from a char* type variable a. The variables **length** and **p** are data members of the class **string**. Once this constructor has been defined in the string class it can be used for conversion from char* type to string type. Example:

```
string s1, s2;
char* namel = "IBMPC”;
char* name2 = “Apple Computer”;
s1 = string (name1);
s2 = name2;
```

The statement

```
s1= string(namel);
```

first converts **namel** from **char*** type to string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

also does the same job by invoking the constructor implicitly.

Class to Basic Type

C++ allows us to define an overloaded *casting operator* that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function usually referred to as a *conversion function*, is:

```
operator typenane()
{
        ........
        ........ (Function statetnents)
        ........
}
```

This function converts a class type data to *typename* . For example the **operator double()**converts a class object to type double, the **operator int()** converts a class type object to type **int**, and so on.Consider the following conversion function:

```
vector :: operator double()
{
double sum = 0;
for {int 1=0: i<size; i++)
sum = sum + v[i] * w[i];
return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator **double()** can be used as follows:

```
double length = double(V1);
        or
double length = VI;
```

where VI is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

One Class to Another Class Type

There are situations where we would like to convert one class type data to another class type. The previous methods are not suitable to convert one class to another class type. Here a new technique is used for this purpose. Example:

```
objX = objY; // objects of different types
```

objX is an object of class X and objY is an object of class Y. The class Y type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from class Y to class X, Y is known as the source class and X is known as the destination class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

The casting operator function, **operator typename(),** converts the class object of which it is a member to *type name*. The *type name* may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, *typename* refers to the destination class, Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves an instruction for converting the argument's type, to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for conversion. This makes it necessary that the conversion constructor be placed in the destination class.

Unit IV – Inheritance

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called **inheritance** (or *derivation*). The old class is referred to as the **base class** and the new one is called the **derived class** or **subclass**.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called **single inheritance** and one with several base classes is called **multiple inheritance**. On the other hand, the traits of one class may be inherited by more than one class. This process is known **as hierarchical inheritance**. The mechanism of deriving a class from another 'derived class' is known as **multilevel inheritance**.

Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details, The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
        ..........//
        ..........// members of derived class
        .........//
};
```

The colon indicates that the derived-class-name is derived from the base class name. The visibility mode is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived* or *publicly derived*.

Examples:

```
        class ABC: private XYZ        // private derivation
        {
                members of ABC
        };

        class ABC: public XYZ         // public derivation
        {
                members of ABC
        };

        class ABC: XYZ                // private derivation by default
        {
                members of ABC
        };
```

When a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can, only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is publicly inherited, 'public members’ of the base class become 'public members’ of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Forms of Inheritance

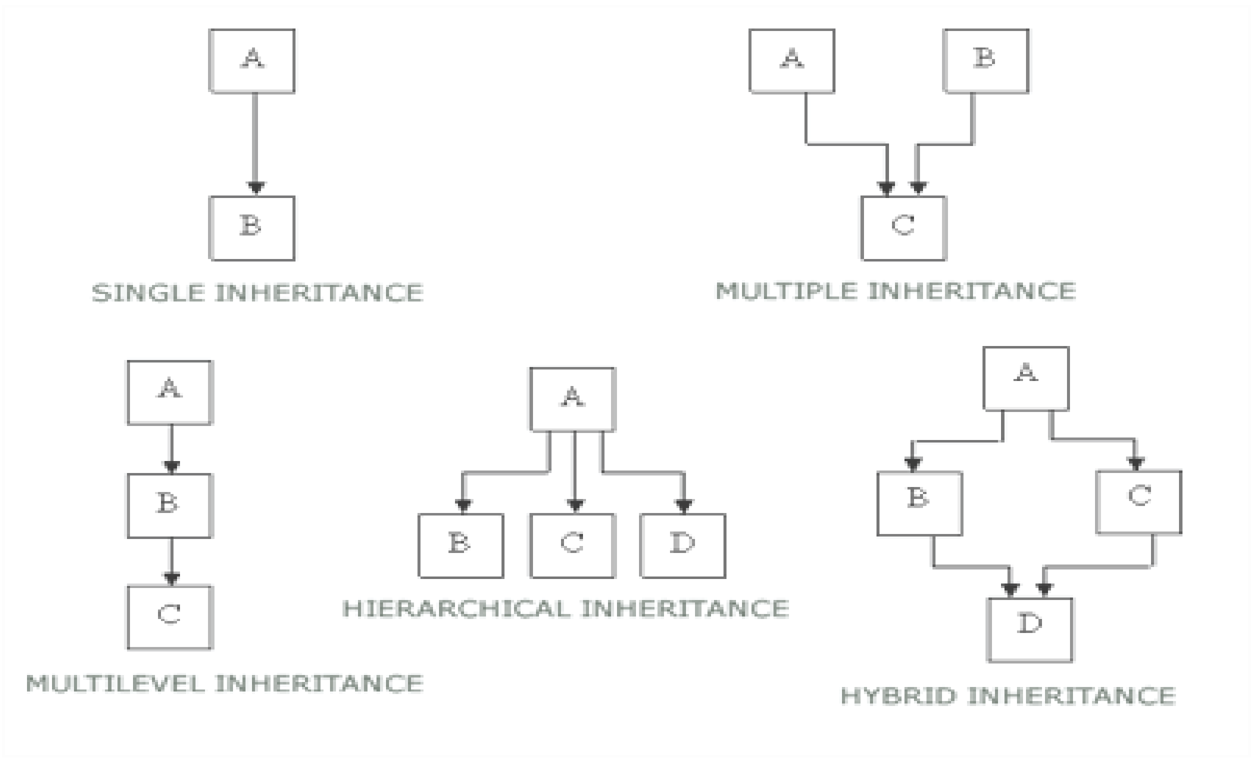
Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)

Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.



1. Single Inheritance

In single inheritance there are two classes combine together for object creation. Here the newly formed class is known as derived class and the existing class is base class. The properties of base class shares to the derived class and the derived class object can access both classes' functionalities, if this is in a public inheritance. Example,

```
class B
{
        int i; // private; not inheritable
protected:
        int j;
public:
        int b; // public; ready for inheritance
        void get_ab();
        int get_a(void);
        void sho_a(void);
};

class D : public B // public derivation
{
        int a;
publc:
        void mul (void);
        void display (void);
};
```

The class D is a public derivation of the base class B. Therefore, D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is also a public member of the derived class D, The private members of B cannot be inherited.

In private derivation, the public members of the base class become private members of the derived class. Therefore, the object of D cannot have direct access to the public member functions of B.

Visibility Modifier 'protected'

C++ provides a third visibility modifier, **protected**, which serve a limited purpose in inheritance. A member declared as **protected** is accessible by the member functions within its class and any class immediately derived from it, if it is in private inheritance. Otherwise, it can access to all classes inherited in the inheritance hierarchy. It cannot be accessed by the functions outside the classes.

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited).

2. Multilevel Inheritance

In multilevel inheritance the first level derived class is again derived to another derived class and this process is continuing to find the final derived class. The object of this derived class can access all the previous classes' properties. The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

A derived class with multilevel inheritance is declared as follows:

```
class A { ………… };                //Base class
class B:  public A { …………}       // B derived from A
class, C: public B  {………....}    //C derived from B
```

This process can be extended to any number of levels.

Assume that the test results of a batch of students are stored in three different classes. Class **student** stores the roll-number, class **test** stores the marks obtained in two subjects and class **result** contains the total marks, obtained in the test. The class **result** can inherit the details of the marks obtained in the **test** and the roll-number of students through multilevel inheritance1. Example:

```
class student
{
        protected:
                int roll_number;
        public:
                void get_number(int );
                void put_number (void) ;
}:
class test : public student // First level derivation
{
        protected:
                float subl;
                float sub2;
        public:
                void get_marks {float , float);
                void put_marks(void);
};
class result : public test // Second level derivation
{
                float total; // private by default
        public:
                void display(void);
};
```

The class result, after inheritance from 'grandfather' through 'father', would contain the following members:

```
private:
        float total;            //own member
protected:
        int roll_number;        //inherited from student via test
        float subl;             //inherited from test
        float sub2;             //inherited from test
public:
void get_number(int);           // from student via test
void put_number (void);         // from student via test
void get_marks (f1oat, float);// from test
void put_marks (void);          // from test
void display(void);             // own number
```

3. Multiple Inheritance

A class can inherit the attributes of two or more classes is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another. Here, more than one base classes and one derived class combine together as an inheritance type.

```
class M
{
        protected:
                int m;
        public:
                void get m(int);
};
class N
{
        protected:
                int n;
        public:
        void get_n(int);
};
class P : public M, public N
```

```
        public:
                void display (void);
};
```

4. Hierarchical Inheritance

Inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.
As an example, it is a hierarchical classification at students in a university. Another example could be the classification of ‘account.' in a commercial bank. All the students have certain things in common and, similarly, all the accounts possess certain common features.

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

Example,

```
class Student
{
        ……….
        ………. //class members
        ……….
};
class  Arts : public Student

{
        ……….
        ………. //class members
        ……….
};
class  Medical : public Student

{
        ……….
        ………. //class members
        ……….
};
```

```
Class Account

{

……………

};

Class Savings_Account: public Account

{

……………..

};

Class Current_Account: public  Account

{

};
```

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

5. Hybrid Inheritance

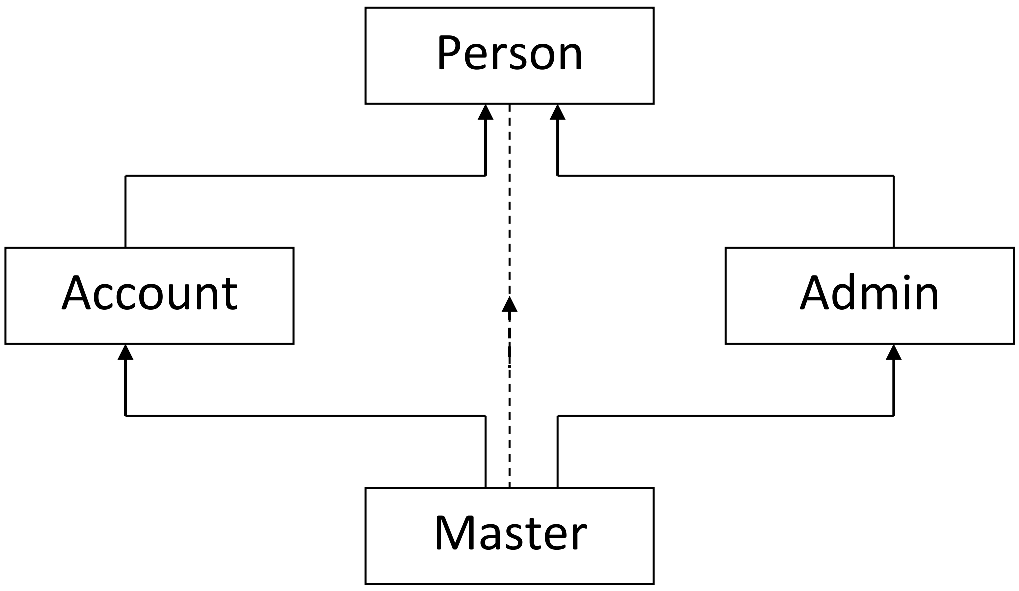
There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be specified as,

```
Class student
{
………..//class members
};
Class sports
{
………..//class members

};
Class test : public student
{
………..//class members

};
Class result : public test, public sports
{

………..//class members

};
```

Multipath Inheritance – C++.

When a class is derived from two or more classes, which are derived from the same base class is known as multipath inheritance. Multipath inheritance can consists many types like single, multiple, multilevel and hierarchical etc.

Example: class Account and Admin derived from Person class and class Master derived from both Account and Admin class.



Use of virtual in multipath inheritance

- In case of multipath inheritance derived class object will have more than one copy of root base class through various paths. To avoid this intermediate classes are declared as virtual classes.
- With reference to above diagram, object of master will have double copy of person through classes admin and account.
- By declaring admin and account as virtual, only one copy of the base class person will be created for the object of class master.
- These classes are known as virtual base classes.

```
//Program to create classes and perform multipath inheritance among them
#include <iostream.h>
#include <conio.h>
class person
{
public: char name[100]; int code;
void input()
{ cout<<"\nEnter the name of the person : "; cin>>name;
cout<<endl<<"Enter the code of the person : "; cin>>code; }
void display()
{ cout<<endl<<"Name of the person : "<<name; cout<<endl<<"Code of the person : "<<code; }
};

class account:virtual public person
{
public: float pay;
void getpay()
{ cout<<endl<<"Enter the pay : "; cin>>pay; }
void display()
{ cout<<endl<<"Pay : "<<pay; }
};


class admin:virtual public person
{
public: int experience;
void getexp()
{
cout<<endl<<"Enter the experience : "; cin>>experience; }
void display()
{
cout<<endl<<"Experience : "<<experience; }
};
class master:publicaccount,public admin
{
```

- public: char n[100];

```
void gettotal()
{
cout<<endl<<"Enter the company name : "; cin>>n; }
void display()
{
cout<<endl<<"Company name : "<<n; }
};

void main()
{  master  m1;    m1.input();    m1.getpay();    m1.getexp();    m1.gettotal();  m1.person::display();
m1.account::display();  m1.admin::display();  m1.display(); getch();
}
```

Output

```
Enter the name of the person : Sheetal

Enter the code of the person : 1768

Enter the pay : 50000

Enter the experience : 1

Enter the company name : SamComputers

Name of the person : Sheetal
Code of the person : 1768
Pay : 50000
Experience : 1
Company name : SamComputers
```

Virtual Base Classes

The need of situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. The 'child' has two direct base classes 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly from ‘child’ to ‘grandparent’. The 'grandparent' is sometimes referred to as indirect base class. Its code is given below:

```
class  A                                //grandparent
{
        ….......
        ……..
};
class Bl : virtual public A             // parent1
{
        ……..
        ……..
};
class B2 : public virtual A             // parent2
```

```
{
          ….......
          ……..
};
class C : public B1, public B2            // child
{
          ……..                    //only one copy of A
          ……..                    // will be inherited
};
```

Inheritance by the 'child' as might pose some problems. All the public and protected members of ‘grandparent' are inherited into 'child' twice, first via parent1 and again via 'parent2', This means, 'child' would have *duplicate* sets of the members inherited from 'grandparent'. This Introduces *ambiguity* and should be avoided.

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

When a class is made a **virtual** base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

Abstract Classes

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development find provides a base upon which other classes may be built. In the previous example, the student class is an abstract class since it was not used to create any objects.

Constructors in Derived Classes

The constructors play an important role in initializing objects. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more argument, then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In ease of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is:

```
Derived-constructor (Arglist1, Arglist2, …… ArglistN, Arglist(D)) : base1(arglist1), base2(arglist2) ... baseN(arglistN)
{
        Body of derived constructor using Arglist(D)
}
```

The header line of derived Constructor function contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the *derived constructor* and the second part lists the function calls to the base constructors.

basel(*arglistl*), *base2*(*arglist2*) ... are function calls to base constructors *base1*(), *base2*(), ….. and therefore *arglistl*, *arglist2* ... etc. represent the actual parameters that are passed to the base constructors. *Arglistl* through *ArglistN* are the argument declarations for base constructors base1 through *baseN*. *ArglistD* provides the parameters that are necessary to initialise the members of the derived class.

Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another by using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
        classalpha {....};
        class beta { …};
        class gamma
        {
                alpha a;        //a is an object of alpha class
                beta b;         //b is an object of beta class
                ……..
                ……..
        };
```

All objects of gamma class will contain the objects a and b. This kind of relationship is called *containership* or *nesting*, Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
        ……..
        ……..
        alpha a;        //a is object of alpha
        beta b;         // b is object of beta
public:
        gamma(arglist) :a(arglistl), b(arglist2)
        {
                // constructor body
        };
```

arglist is the list of arguments that is to be supplied when a gamma object is defined. These parameters are used for initializing the members of gamma, *arglistl* is the argument list for the constructor of **a** and *arglist2* is the argument list for the constructor of **b**. *arglistl* and *arglist2* may or may not use the arguments from *arglist*. Remember, **a**(*arglist*) and **b**(*arglist2*) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
        gamma (int x, int y, float z) : a(x), b(x,z)
        {
                Assignment section (for ordinary other members)
        }
```

We can use as many member objects as are required in a class. For each member object we add a constructor call in the initialized list. The constructors of the member objects are called in the order in which they are declared in the nested class.

Unit V – Pointer Virtual Functions and Polymorphism, Working with Files

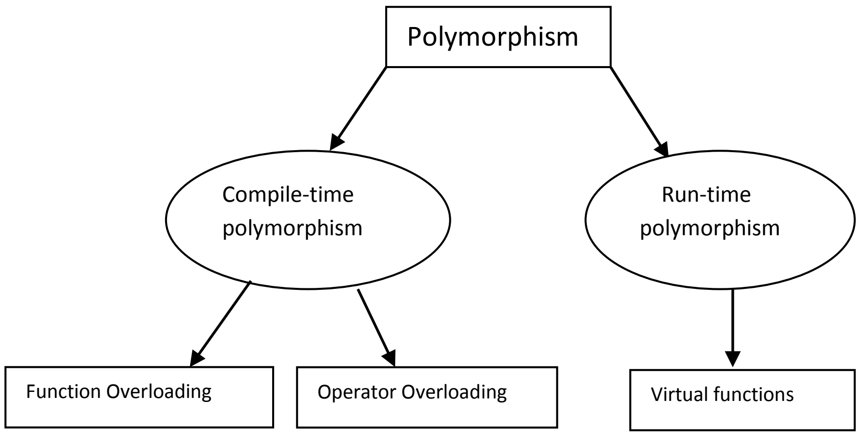
Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. The concept of polymorphism is implemented using the overloaded function and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known as *compile time polymorphism*, early binding simply means that an object is bound to its function call at compile time.

Now consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
        int x;
public:
        void show() { ......}     //show in base class
};
class B : class A
{
        int y;
public:
        void show() { ......}     //show() in derived class
};
```

How do we use the member function show() to print the values of objects of both the classes A and B? Since the prototype of show() is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. C++ supports a mechanism known as *virtual function* to achieve run time polymorphism.



At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding*. It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time. Dynamic binding is one of the powerful features of C++, This requires the use of pointers to objects.

Pointers

Pointer is one of the key aspects of C++ language similar to that of C. Pointers offer a unique approach to handle data in C and C++. A pointer is a derived date type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data. Like C, a pointer variable cad also refer to (or point to) another pointer in C++. However, it often points, to a data variable. Pointers provide an alternative approach to access other data objects.

Pointers to Objects

A pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

Where **item** is a class and **x** is an object defined to be of type **item**. Similarly we can define a pointer **it_ptr** of type **item** as follows:

```
item *it_ptr;
```

Object pointers are useful in creating object at run time. We can also use an object pointer to access the public members of an object. Consider a class **item** defined as follows:

```
class item
{
        int code; float price;
public:
        void getdata(int a, float b)
        {
                Code = a;
                price = b;
        }
        void show(void)
        {
                cout « "Code :" « code << "\n" « "Price: " « price « “\n\n”;
        }
};
```

Let us declare an **item** variable x and a pointer **ptr** to x as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of x.

We can refer to the member functions of **item** in two ways, one by using the **dot operator** and the object, and another by using the arrow operator and the object pointer. The statements

```
x.getdata(100,75.50);
x.show();
```

are equivalent to

```
ptr->getdata(100, 75.50);
ptp ->show();
```

Since *ptr is an alias of x, we can also use the following method:

```
(*ptr).show();
```

The parentheses are necessary because the dot operator has higher precedence than the*indirection operator* (*.). We can also create the objects using pointers and **new** operator as follows:

```
item *ptr =new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr.** Then **ptr** can be used to refer to the members as shown below:

```
ptr ->show();
```

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created. We can also create an array of objects using pointers. For example, the statement

```
item *ptr = new 1tem[10]; // array of 10 objects
```

creates memory space for an array of 10 objects of **item**. Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

this **Pointer**

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which this function was called. For example, the function call A.max() will set the pointer **this** to the address of the object A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the member functions. Consider the following simple example:

```
class ABC
{
int a;
........
........
};
```

The private variable ‘a’ can be used directly inside a member function, like

```
a = 123;
```

We can also use the following statement to do the same job:

```
this->a  = 123:
```

Since C++ permits the use of shorthand form a = 123, we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator in overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**. One important application of the pointer **this** is to return the object it points to. For example, the statement

*return *this;*

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result, Example;

```
person & person greater(person & x)
{
        if x.age > age
        return x;         // argument object
        else
        return *this: // invoking object
}
```

Suppose we invoke this function by the call

```
Max =  A.greater(B);
```

The function will return the object B (argument object) if the age of the person B is greater than that of A, otherwise, it will return the object A (invoking object) using the pointer **this**. Remember, the dereference operator * produced the contents at the address contained in the pointer.

Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from **B**, then a pointer declared as a pointer to B can also be a pointer to **D**. Consider the Following declarations:

```
B *cptr;                // pointer to class B type variable
B b;                   //base object
D d                     // derived object
cptr = &b;             // cptr points to object b
```

We can make **cptr** to point to the object d as follows:

```
Cptr = &d;      // cptr points to object d
```

This is perfectly valid with C++ because **d** is an abject derived from the class **B**.

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

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

Virtual Functions

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism in therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. Polymorphism is achieved using what is known as ‘virtual' functions.

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration, When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. The following segment illustrate this point.

```
class Base
{       public:
                void display() {cout<<”\nDisplay Base”; }
                virtual void show() {cout<<”\n Show Base”; }
};

class Derived : public Base
{       public:
                void display() {cout<<”\nDisplay Derived”; }
                void show() {cout<<”\nShow Derived”; }
};

void main()
{
        Base B;         Derived D;      Base *bptr;
        cout <<“\n bptr points to Base ”;               bptr = &B;
        bptr->display(); //calls Base version
        bptr->show(); //calls Base version

        cout<<”\n\n bptr points to Derived ”; bptr = &D;

        bptr -> display();      //calls Base version

        bptr -> show();         //calls Derived version
}
```

Output:

```
bptr points to Base
Display Base
Show Base
bptr points to Derived
display Base
show Derived
```

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should, observe some basic rules that satisfy the compiler requirements:

1.The virtual functions must be members of some class.

2. They cannot be static members.

3. They are accessed by using object pointers.

4. A virtual function can be a friend of another class.

5. A virtual function in a base class must be defined, even though it may not be used.

6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them, as overloaded functions, and the virtual function mechanism is ignored.

7. We cannot have virtual constructors, but we can have virtual destructors.

8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.

9. When a base pointer points to a derived class, incrementing: or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type . Therefore, we should not use this method to move the pointer to the next object.

10, If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class, in such eases the calls will invoke the base function.

Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder*. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

virtual void display() = 0;

Such functions arc called *pure virtual functions*. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Working with Files

Many real -life problems handle large volumes of data and, in such situations, we need to use some devices such as pen drives or hard disk to store the data. The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files. A program typically involves either or both of the following kinds of data communication:

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

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams, as an interface between the programs and the files. The stream that supplies data to them program is known as *input stream* and the one that receives data from the program is known as *output stream*. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file.

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

File Stream Classes

The I/O system of'C + + contains a set of classes that define the file handling methods. These include **ifstream, ofstrcam** and **fstream.** These classes are derived from **fstreambase** and from the corresponding *iostream* class as shown below. These classes, designed to manage the disk files, are declared in *fstream* and therefore we must include this file in any program that uses files.

Opening and Closing a File

If we want to use a disk file, we need to decide the following things about the file and its intended use:

1. Suitable name for the file,
2. Data type and structure.
3. Purpose,
4. Opening method.

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension. For opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes **ifstream**, **ofstream** and **fstream** that are contained in the header file **fstream**. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class,

2 Using the member function open() of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening Files Using Constructor

We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialise the file abject with the desired filename.

For example, the following statement opens a file named "results" for output:

```
ofstream outfi1e ( "results");   // output only
```

This creates **outfile** as an ofstream object that manages the output stream. Similarly, the following statement declares *inflle* as an **ifstream** object and attaches it to the file data for reading (input).

```
ifstream infi1e( "data" ) ; // input only
```

The program may contain statements like:

```
outfile <<"TOTAL”;
outfile <<sum;
infile >>number;
infile >>string;
```

Opening Files Using open()

Aa stated earlier, the function open() can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

file- stream- class stream-object;
stream -object.open ("filename");

Example:

```
ofstream outflie;             // Create stream (for output)
outfile.open(“DATA1”);        //Connect stream to DATA1
.........
........
outfile.close();              // Disconnect stream from DATA1
outfile.open(“ DATA2”);       //Connect stream to DATA2
.........
.........
Outfile.close ();             // Disconnect stream from DATA2
..........
..........
```

File Opening modes

We have used **ifstream** and **ofstream** constructors and the function **open**() to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename. However, these functions can take two arguments, the second one for specifying the *file mode*. The general form of the function open() with two arguments is:

```
        Stream -object.open(" filename", mode);
```

The second argument mode (called file mode parameter) specifies the purpose for which the file is opened.

File Mode Parameters

Parameter	*Meaning*
Ios:: app	Append to end of file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists.

File Pointers and Their Manipulations

Each file has two associated pointers known as the *file pointers*. One of them is called the input printer (or get pointer) and the other is called the output pointer (or put pointer}. We can use these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place, the appropriate pointer is automatically advanced,

- **Default Actions**

When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and the output painter is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to add more data, the file is opened in *append* mode. Thin moves the output pointer to the end of the file (i,e, the end of the existing contents).

The file stream classes support the following functions to manage such situations:

• seekg()	Moves get pointer (input) to a specified location.
• seekp()	Moves put pointer(output) to a specified Location.
• tellg ()	Gives the current position of the get pointer,
• tellp()	Gives the current position of the put pointer.

Sequential Input and Output Operations

The file stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, **put**() and **get**(), are designed for handling a single character at a time. Another pair of functions, **write()** and **read**(), are designed to write and read blocks of binary data.

The functions **write**() and **read**(), unlike the functions **put**() and **get**(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. The binary format is more accurate for storing the numbers as they are stored in the exact internal representation. There are no conversions while saving the data and therefore saving much faster.

The binary input and output functions take the following form:

infile.read((*char) &V, sizeof(V));
outfile.write((*char) &V, sizeof(V));

Example:

```
#include<iostream>
#include<fstream>
#include<cstdio>
using namespace std;

class Student
{
    int admno;
    char name[50];
public:
    void setData()
    {
        cout << "\nEnter admission no. ";
        cin >> admno;
        cout << "Enter name of student ";
        cin.getline(name,50);
    }

    void showData()
    {
        cout << "\nAdmission no. : " << admno;
        cout << "\nStudent Name : " << name;
    }

    int retAdmno()
    {
        return admno;
    }
};

/*
* function to write in a binary file.
*/

void write_record()
{
    ofstream outFile;
    outFile.open("student.dat", ios::binary | ios::app);

    Student obj;
    obj.setData();
    outFile.write((char*)&obj, sizeof(obj));

    outFile.close();
}

/*
* function to display records of file
*/


void display()
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);

    Student obj;

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        obj.showData();
    }

    inFile.close();
}

/*
* function to search and display from binary file
*/

void search(int n)
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);

    Student obj;

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            obj.showData();
        }
    }

    inFile.close();
}

/*
* function to delete a record
*/

void delete_record(int n)
{
    Student obj;
    ifstream inFile;
    inFile.open("student.dat", ios::binary);

    ofstream outFile;
    outFile.open("temp.dat", ios::out | ios::binary);

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() != n)
        {
            outFile.write((char*)&obj, sizeof(obj));
        }
    }

    inFile.close();
    outFile.close();

    remove("student.dat");
    rename("temp.dat", "student.dat");
}

/*
* function to modify a record
*/

void modify_record(int n)
{
    fstream file;
    file.open("student.dat",ios::in | ios::out);

    Student obj;

    while(file.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            cout << "\nEnter the new details of student";
            obj.setData();

            int pos = -1 * sizeof(obj);
            file.seekp(pos, ios::cur);

            file.write((char*)&obj, sizeof(obj));
        }
    }

    file.close();
}

int main()
{
    //Store 4 records in file
    for(int i = 1; i <= 4; i++)
       write_record();

    //Display all records
    cout << "\nList of records";
    display();

    //Search record
    cout << "\nSearch result";
    search(100);

    //Delete record
    delete_record(100);
    cout << "\nRecord Deleted";

    //Modify record
    cout << "\nModify Record 101 ";
    modify_record(101);

    return 0;
}
```