6... Polymorphism

Objectives...

- To study about Compile time and run time polymorphism.
- To study function overloading.
- To learn about operator overloading.

6.1 INTRODUCTION [S-18, 23, W-18]

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:
  1. **Compile time polymorphism:** This is also called as **early** or **static** binding. Selection of an appropriate function for a particular call at the compile time itself. For example, function overloading and operator overloading.
  2. **Run-time polymorphism:** This is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.

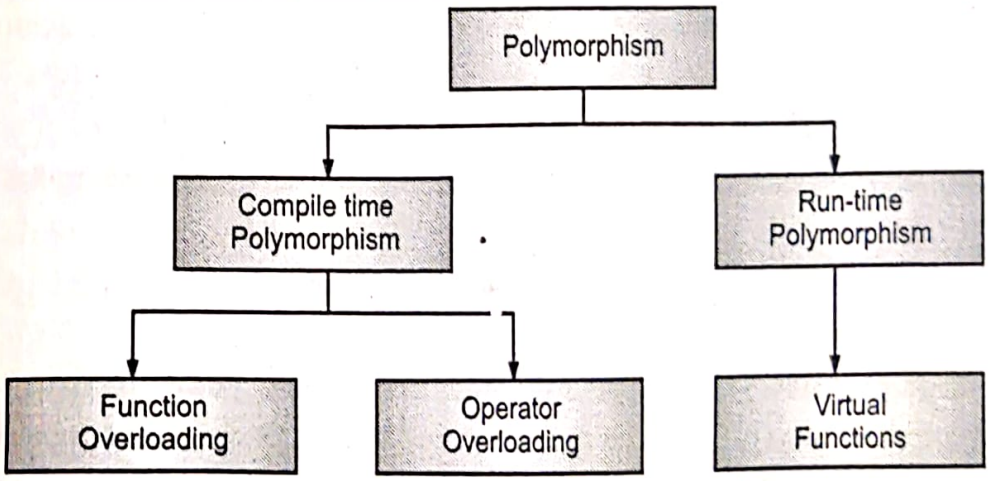


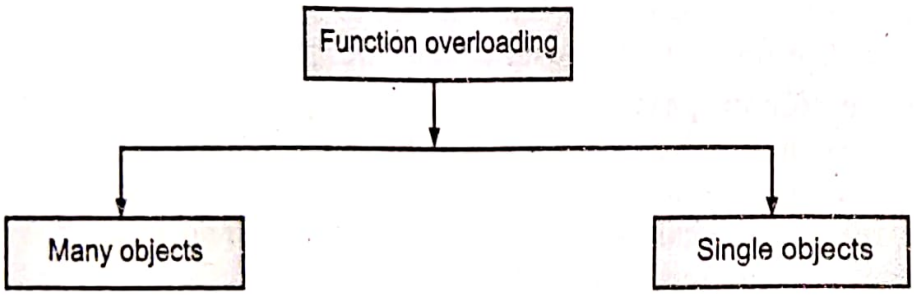
Fig. 6.1: Types of Polymorphism

6.2 COMPILE TIME POLYMORPHISM [S-19, 22]

- Polymorphism, in C++, is implemented through overloaded functions and overloaded operators.
- Function Overloading is also referred to as functional polymorphism. The same function can perform a wide variety of tasks.
- The same function can handle different data types. When many functions with the same name but different argument lists are defined, then the function to be invoked corresponding to a function call is known during compile time.
- When the source code is compiled, the functions to be invoked are bound to the compiler during compile time, as to invoke which function depending upon the type and number of arguments. Such a phenomenon is referred to early binding, static linking or compile time polymorphism.

6.2.1 Function Overloading [S-19, W-22]

- C++ provides the facility of function overloading in which we can define multiple functions with the same name but the types of argument are different.
- For example, we can have a function add with two floating point numbers as argument, another function add with 3 integers, one add function with 4 doubles and so on. But function name is similar. This is called as **Function Overloading**.
- An overloaded function, is a function with the same name as another function, but with different parameter types.
- The function overloading can be performed using single and many objects. This is shown in following Fig. 6.2.



- Same function name with different argument list.
- Different objects are created.
- A call is given to different arg-list functions with each object.

- Same function name with different argument list.
- One object is created.
- A call is given to different arg-list function with one object.

Fig. 6.2: Function Overloading

- **Rules for overloaded functions:**
  1. The argument list of each of the function instances must be different.
  2. The compiler does not use the return type of the function to distinguish between function instances.
- The Program 6.1 explains the concept of function overloading with many objects and Program 6.2 explains the concept with one (single) object.

Program 6.1: Program for function overloading with many objects.

```
#include<iostream.h>
class test
{
    public:
    void addnum (int p, int q, int r)
     {
        cout<<"sum is"<<p+q+r<<"\n";
     }
     void addnum (float p, float q)
     {
        cout<<"sum is"<<p + q <<"\n";
     }
     void addnum(int p, double q)
     {
        cout<<"sum is"<<p + q<<"\n";
     }
};
int main()
{
    test ob1, ob2, ob3;
    ob1.addnum (5, 8, 10);
    ob2.addnum (2.5, 3.6);
    ob3.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

- The difference in return type is not a consideration for function overloading.

Program 6.2: Program for function overloading with single object.

```
#include<iostream.h>
class test
{
    public:
    void addnum(int p, int q, int r)
    {
     cout<<"sum is"<<p+q+r<<"\n";
    }
```

```
    void addnum (float p, float q)
    {
     cout<<"sum is"<<p+q<<"\n";
    }
    void addnum(int p, double q)
    {
    cout<<"sum is"<<p+q<<"\n";
    }
};
int main()
{
    test ob1;
    ob1.addnum (5, 8, 10);
    ob1.addnum (2.5, 3.6);
    ob1.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

6.2.2 Operator Overloading [S-22, 23]

- Operator overloading is closely related to function overloading.
- In C++, you can overload most operators so that they perform special operations relative to classes that you create.
- Actually, C++ tries to make user defined data types behave in much the same way as 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'**.
- Operators can be overloaded by creating **'operator functions'**.
- An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work.
- An operator function is created using the keyword **'operator'**.
- A operator function's general form is:

```
return_type class_name:: operator op (arg-list)
{
    // operations
}
```

Where, `return-type` is the type of value returned by the specified operation and `op` is the operator being overloaded.

- For example, if you are overloading the / operator, use 'operator /'. When a unary operator is overloaded, arg-list is empty.
- When a binary operator is overloaded, arg-list will contain one parameter.
- The process of overloading involves the following steps:
  1. First, create a class that defines the data type that is to be used in the overloading operation.
  2. Declare the operator function operator op ( ) 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.
- Almost all C++ operators can be overloaded.
- Operators that can be overloaded except the following:

  . Member access operator

  .* Pointer to member operator

  :: Scope resolution operator

  ?: Conditional operator

  sizeof sizeofoperator.
- The *precedence* of an operator cannot be changed by overloading. This can lead to awkward situations in which an operator is overloaded in a manner for which its fixed precedence is inappropriate. However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- The associativity of an operator cannot be changed by overloading.
- It is not possible to change "*arity*" of an operator (i.e. the number of operands an operator takes): Overloaded *unary operators* remain as unary operators; overloaded *binary operators* remain as binary operators. C++ only ternary operator (?:) cannot be overloaded.
- The meaning of an operator works on objects of built-in types cannot be changed by operator overloading. For example, change the meaning of how + adds two integers.
- Operator overloading only works with objects of user defined types or with a mixture of an object of a user defined type and an object of a built-in type.
- Overloading an assignment operator with an addition operator to allow statements like:

```
object 2 = object 2 + object 1
```

  Does not imply that + = operator is also overloaded to allow statements such as,

```
object 2 + = object 1;
```

- Such behaviour can be achieved by *explicitly overloading* the + = operator for that class.
- Different types of operators are:

  1. Arithmetic operators:

  +, −, /, *, %, =, ++, − −, % is called modulus operator used for getting remainder of division.

2. **Relational operators:**
   These operators are used for checking for condition. Here, L.H.S. value is compared with R.H.S. Operators are: <, >, <=, >=, = = , !=
3. **Logical operators:**
   These operators work similarly to logic gates AND, OR, NOT, XOR. Operators are: &&, || , !.
4. **Bitwise operators:**
   These operators operate on every bit of a byte. Operators are: &, |

- According to the operands required for operation all operators are divided into three categories:

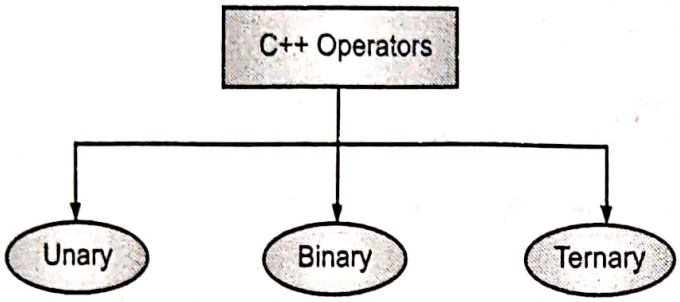


Fig. 6.3: C++ Operator

1. **Unary:**
   It requires only one operand to perform operation. For example: `i++`
   Here, i is the single operand to perform increment operation.
2. **Binary:**
   It requires two operands to perform operation. For example: `a + b;`
   Here, a and b are two operands to perform addition operation.
3. **Ternary operator:**
   It requires three operands to perform operation. Ternary operator is ?,:
   For example: `x > 5? y = 2: y = 1;`
   This line compares x value. If x > 5, y = 2. If x is not greater than y = 1.

- Overloading means assigning more than one meaning to operator.
- For example, + sign can be used for:
  1. Addition of two integers
  2. Addition of two complex numbers.
  3. Addition of two characters.
- **Steps for operator overloading are given below:**
  1. Create a member function under a public section.
  2. The member function is called as operator overloaded function so, syntax of declaration than other normal member function.

     Syntax:

```
return_type operator op( )
{
   //details of operation
}
```

     Here, operator is a keyword which tells it is operator overloading function and op is any operator, For example, +.

3. When we want to call operator overloading function the syntax is:

```
operator sign object_name;
```

For example: For calling + sign's operator overloaded function we will use, `+a1;`

Here, `a1` is the object of a class where operator overloading function is declared.

6.2.3 Rules for Overloading Operators [W-18]

- Although it looks simple to redefine the operators, there are certain restrictions and/or 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 we cannot redefine the plus (+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators that cannot be overridden.
5. There are some operators that cannot be overloaded such as .*,:: etc.
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.
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 +, −, *, / must explicitly return a value. They must not attempt to change their own arguments.
10. We cannot use friend functions to overload certain operators, which are listed below. However, member functions can be used to overload them.

    = Assignment operator
    () Function call operator
    [] Subscripting operator
    −> Class member access operator.

Where, a friend cannot be used.

6.2.4 Operator Overloading Unary and Binary

1. Overloading Unary Operator:

- The operators can be overloaded using two different functions i.e. through member functions and friend functions.
- A unary operator overloaded using a member function takes no argument, whereas an overloaded unary operator declared as friend function takes one argument.

- The unary operators that can be overloaded are shown below.

Table 6.1: Unary Operator

Operators	Meaning
->	Indirect member operator
!	Logical negation
*	Pointer reference
&	Address operator
~	Ones complement
->*	Indirect pointer to member
+	Addition
-	Subtraction
++	Incrementer
--	Decrementer
-	Unary minus.

- First we will consider the unary minus (-) operator. A minus operator, when used as a unary, takes just one operand.
- We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable.
- The unary minus when applied to an object should change the sign of each of its data items.

Program 6.3: Program for how the unary minus operator is overloaded.

```
#include<iostream.h>
class space
{
    int x;
    int y;
    int z;
    public:
        void getdata (int a, int b, int c);
        void display (void);
        void operator - ( );  // overload unary minus
};
void space:: getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
```

```
void space:: display (void)
{
   cout << x << " ";
   cout << y << " ";
   cout << z << "\n";
}
void space:: operator - ( ) // defining operator - ( )
{
   x = - x;
   y = - y;
   z = - z;
}
int main ( )
{  space s;
   s.getdata (10, -20, 30);
   cout << "s:  before overloading:";
   s.display ( );
   - s;                    // activates operator - ( )
   cout << "s: After overloading:";
   s.display ( );
}
```

Output:

```
s: before overloading:10 -20 30
s: After overloading:-10 20 -30
```

- Note that, the function operator – ( ) takes no argument. Then what does this operator function do? It changes the sign of data members of the objects.
- Since, this function is a member function of the same class; it can directly access the members of the object which activated it.
- Remember a statement like:

```
s2 = - s1;
```

Will not work because, the function operator – ( ) does not return any value.

- It can work if the function is modified to return an object.
- It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator - (space & s);   // declaration
void operator - (space & S)   // definition
{
   s.x = -s.x;
   s.y = -s.y;
   s.z = -s.z;
}
```

- Note that, the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator – ().
- Therefore, the changes made inside the operator function will not reflect in the called object.

Overloading of Increment Operator:

- We know that C++ supports the operator that is used for incrementing and decrementing by 1. These operators can be used either prefix or postfix.
- Generally, overloading of these operators cannot be distinguished between prefix or postfix operation. But whenever a postfix operation is overloaded, it takes a single argument along with a member function of a class object.

Program 6.4: Program to generate Fibonacci series by overloading a prefix operator.

```
#include<iostream.h>
class fibo
{
   private:
      int fib0, fib1, fib2;
   public:
      fibo(); // default constructor
      void operator ++();  // overloading declaration
      void disp();
};
fibo:: fibo()
{
   fib0 = 0;
   fib1 = 1;
   fib2 = fib0 + fib1;
}
void fibo:: disp()
{
cout<<fib2<<"\t";
 }
void fibo:: operator ++()
{
   fib0 = fib1;
   fib1 = fib2;
   fib2 = fib0 + fib1;
}
```

```
int main()
{
    fibo ob1;
    int n;
    cout<<"How many numbers you want?:";
    cin>>n;
    for(int i=0;i<=n;++i)
    {
        ob1.disp();
        ++ob1;
    }
}
```

Output:

```
How many numbers you want?:
4
1       2       3       5       8
```

2. Overloading Binary Operator:

- Binary operators are operator which require two operands to perform operation.
- We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator.
- To add two numbers generally we use statement like:

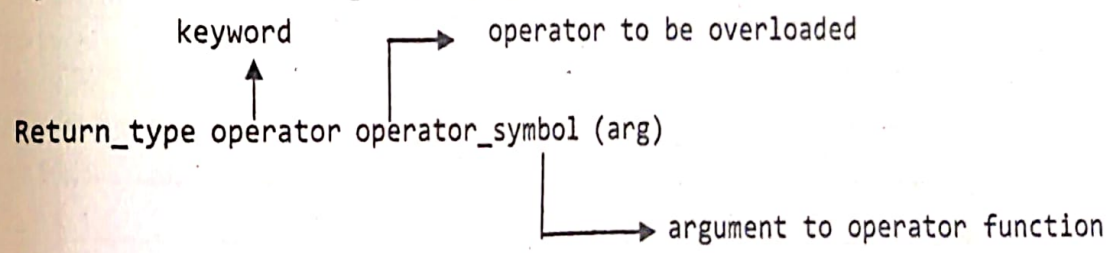
```
C = sum (A, B)      // functional notation
```

- The functional notation can be replaced by a natural looking expression,

```
C = A + B           // arithmetic notation
```

By overloading the + operator using an operator + ( ) function.

- The binary operators overloaded through member function take one argument which is formal.
- The Table 6.2 shows the binary operators which can be overloaded. The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly.
- **The syntax for overloading a binary operator is:**




```
{
    // body of operator function
}
```

Table 6.2: Binary Operators

Operator	Meaning
[ ]	Array element reference
( )	Function call
new	New operator
delete	Delete operator
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal
!=	Not equal
&	Bitwise AND
^	Bitwise XOR
&&	Logical AND
||	Logical OR
=	Assignment
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
+=	Add and assign
-=	Subtract and assign
<<=	Shift left and assign
>>=	Shift right and assign
&=	Bitwise AND and assign
:=	Bitwise OR and assign
^=	Bitwise one's complement and assign

rogram 6.5: Program for binary operator overloading. [S-23]

```
#include<iostream>
using namespace std;
class integer
{
    private:
        int val;
    public:
        integer(); //constructor1
        integer(int one); //constructor2
        integer operator + (integer objb); //operator function
        void disp();
};
integer:: integer()
{
    val = 0;
}
integer::integer(int one)
{
    val = one;
}
integer integer::operator + (integer objb)
{
    integer objsum;
    objsum.val = val + objb.val;
    return(objsum);
}
void integer::disp()
{
    cout<<"value="<<val<<endl;
}
int main()
{
    integer obj1 (11);
    integer obj2 (22);
     integer objsum;
    objsum = obj1 + obj2; //operator overloading
    obj1.disp();
    obj2.disp();
    objsum.disp();
}
```

Output:

```
value = 11
value = 22
value = 33
```

6.2.5 Operator Overloading Using Friend Function

- We have already seen that private members of a class cannot be accessed through outside functions. This is possible just by using friend function concept.
- Friend functions may be used in place of member functions for overloading a binary operator. The difference is friend function requires two arguments to be explicitly passed to it while a member function requires only one.

Overloading Binary Operators using 'friend' Functions:

- The syntax for declaring and defining a friend operator function to overload binary operator is as follows:

```
class class_name
{
    public:
        friend return_type operator op(arg1, arg2);
};
return_type operator op (argl, arg2, … argN)
{
    //function definition ……
}
```

Program 6.6: Program for overloading binary operator using friend function.

```
#include<iostream.h>
class binopr
{
    private:
        int x;
    public:
        binopr() { } // constructor
        binopr (int); // parameterized constructor
        void disp();
        friend binopr operator + (binopr,binopr);
};
binopr:: binopr(int a)
{
    x=a;
}
void binopr:: disp()
{
    cout<<x;
}
```

```
// overloading definition with friend function
binopr operator + (binopr P, binopr S)
{
    return (P.x + S.x);
}
int main()
{
    binopr ob1, ob2, ob3;
    ob1 = binopr (15);
    ob2 = binopr (28);
    ob3 = ob1 + ob2;
    ob3.disp();
}
```

Output:

```
43
Press any key to continue . . .
```

- The overloading definition can also be written using following code:

```
binopr operator + (binopr P, binopr S)
{
    S.x = P.x + S.x;
    return(S.x);
}
```

- The implementation of overloading + operator (binary) is shown in Fig. 6.4.

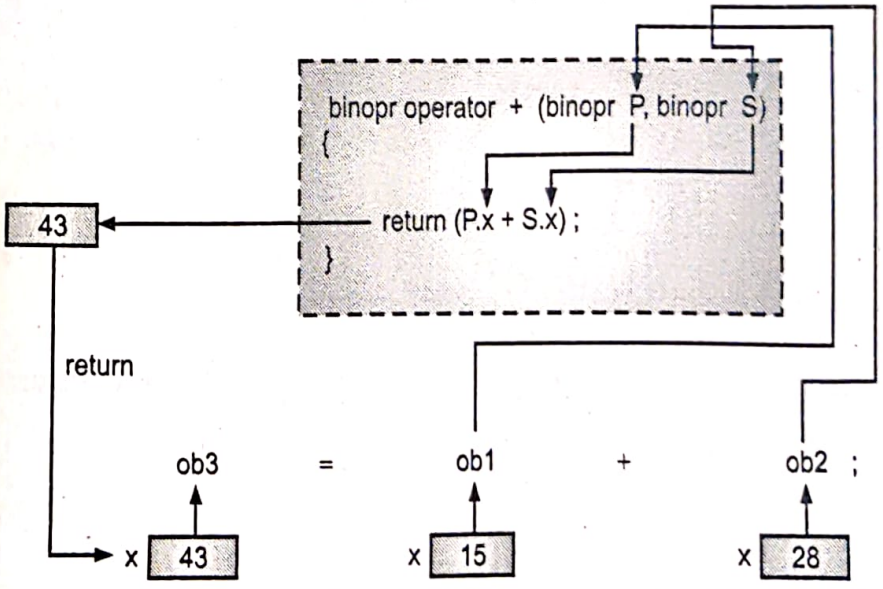


Fig. 6.4: Implementation of overloaded + operator using friend function

Overloading Unary Operators using 'friend' Functions:

- Here, the operator function is not the member function of any class. Hence, in this scenario if an operator has to be overloaded to make it work on a class object, the operator function has to be declared as a friend to that class.
- It is to be noted that, when an operator function is friend to a class, then we can call that function as a "friend operator function."
- The syntax for declaring and defining a friend operator function to overload an unary operator is as follows:

```
class class_name
{
    public:
        friend return_type operator op(arg);
};
return_type operator op(arg)
{
    //function definition .....
}
```

Program 6.7: Program for overloading unary operators using friend function.

```
#include<iostream>
using namespace std;
class OverUnaOpr
{
    private:
        int x;
        int y;
    public:
        friend void operator - (OverUnaOpr &); // prototype
        void display( );
        OverUnaOpr( )// constructor
        {
            x = 6;
            y = 7;
        }
};
void operator - (OverUnaOpr &a) // operator function definition
{
    a.x = -a.x;
    a.y = -a.y;
}
```

```
void OverUnaOpr:: display ( )
{
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}
int main()
{
    OverUnaOpr s;
    s.display();
    -s;          //Overloading unary '-' operator
    s.display();
    return 0;
}
```

Output:

```
x = 6
y = 7
x = - 6
y = - 7
```

6.2.6 Overloading Insertion (<<) and Extraction (>>) Operators [S-22, 23]

- We have already used the objects **cin** and **cout** (pre-defined in iostream file) for the input and output of data of various types. This is possible only because the operators >> and << are overloaded to recognize all the basic C++ types.
- The >> operator is overloaded in the **istream** class and << operator is overloaded in the **ostream** class. The istream class overloads the >> operator for the standard types [int, long, double, float, char, and char * (string)].
- For example, the statement, `cin >> x;` calls the appropriate >> operator function for the istream cin defined in iostream.h. And uses it to direct this input stream into the memory location represented by the variable x.
- Similarly, the ostream class overloads the << operator, which allows the statement, `cout << x;` to send the value of x to ostream cout for output.
- Actually, istream provides the generic code for formatting the data after it is extracted from the input stream. Similarly, ostream provides the generic code for formatting the data before it is inserted to the output stream.

Program 6.8: Accept and display the information using extraction and insertion operators. [S - 19]

```
#include<iostream>
using namespace std;
class info
{
   private:
          int roll;
          char name[20];
```

```
    public:
            info()
            {
               roll = 0;
               name [0] = '0';
            }
            friend istream & operator>>(istream &, info&);
            friend ostream & operator<<(ostream &, info&);
};
istream & operator>>(istream &S, info &d)
{
    cout<<"Enter Rollno";
    S>>d.roll;
    cout<<"Enter name";
    S>>d.name;
    return S;
}
ostream & operator<<(ostream &S, info &d)
{
    S<<d.roll;
    S<<d.name;
    return S;
}
int main ()
{
    info ob1;
    cin>>ob1;
    cout<<ob1;
}
```

Output:

```
Enter rollno: 101
Enter name: Prajakta
101 Prajakta.
```

6.2.7 String Manipulation Using Operator Overloading

- C++ permits us to create own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers.
- Strings can be defined as class objects which can be then manipulated like the built-in types.

- For example:

```
string S1 = string S2 + string S3;
if(string S1 > = string S2)
string = string S1
```

- A typical string class will like this,

```
class string
{
    char * p;    /* pointer to string */
    int len;     /* length to string */
    public:
    /* member function for initialize and manipulate strings */
    .....
    .....
    .....
};
```

Program 6.9: Providing a reversing operator for string class by overloading operator ~.
For example: If string = C++ then output = ++C.

```
#include<iostream>
#include<cstring>
using namespace std;
class String
{
    private:
       char str[40];
    public:
        void getdata()
        {
            cout<<"Enter string to be reversed";
            cin >> str;
        }
        void operator ~ ()
        {
            strrev (str);
        }
        void display()
        {
            cout <<"Reversed string = "<< str;
        }
};
```

```
int main()
{
    String s1;
    s1.getdata(),
    ~ s1;
    s1.display();
    return 0;
}
```

Output:

```
Enter string to be reversed
MAHESH
Reversed string = HSEHAM
Press any key to continue . . .
```

Program 6.10: Overload the operator ! to find out the length of the string. [S-18, 19; W-18, 22]

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class String
{
    private:
        char str[40];
    public:
        void getdata()
        {
            cout<<"Enter the string";
            cin >> str;
        }
        int operator ! ()
        {
            int l;
            l = strlen(str);
            return(l);
        }
        void display();
} s1;
void String:: display()
{
    cout <<"length of string=";
    cout << ! s1;
}
```

```
int main()
{
    s1.getdata();
    s1.display();
    getch();
    return 0;
}
```

Output:

```
Enter the string
MAHESH
length of string=6
Press any key to continue . . .
```

Program 6.11: Write a program to overload the + operator so that two strings can be concatenated. For example: s1 = abc, s2 = pqr then abc + pqr = abcpqr.

```
#include<iostream.h>
#include<conio.h>
class String
{
    char str[20]; //member variable for string input
    public:
        void input() //member function
        {
            cout<<"Enter your string: ";
            cin.getline(str,20);
        }
        void display() //member function for output
        {
            cout<<"String: "<<str;
        }
        String operator+(String s) //overloading
        {
            String obj;
            strcat(str,s.str);
            strcpy(obj.str,str);
            return obj;
        }
};
int main()
{
    String str1,str2,str3; //creating three objects
    str1.input();
    str2.input();
    str3=str1+str2;
    str3.display();
}
```

6.3 RUNTIME POLYMORPHISM [S - 19]

- Run-time polymorphism is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.
- To achieve run-time polymorphism C++ supports a mechanism known as **virtual functions**. Dynamic binding uses the concept of pointers, i.e. it requires to use of pointer to object.

6.3.1 'this' Pointer [S-23]

- Every object in C++ has access to its own address through an important pointer called 'this' pointer.
- In other words, "The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- The 'this' pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a 'this' pointer, because friends are not members of a class. Only member functions have a 'this' pointer.
- 'this' is a pointer that points to the object for which the function was called.
- 'this' pointer is predefined pointer variable within every class.
- 'this' pointer is a pointer pointing to the current object of specified class. "this" is a keyword in C++.
- **Syntax:**

```
class_name * this;
```

- Every object has access to its own address through a pointer called **this**. When a member function is called, 'this' pointer is automatically passed as an implicit argument to that function.
- For example: The function call,

`s.sample();` will set the pointer this to the address of the object `s`.

- One of the important application of "this" pointer is to return the object to which it is pointing. For example the following statement,

```
return * this;
```

Inside a member function will return the object that invokes a member function.

Program 6.12: Write a program to class test having data members as name of a student and marks. Accept and display information by using 'this' pointer.

```
#include<iostream>
using namespace std;
class test
{
   private:
      char name[40];
      float marks;
```

```
    public:
    void get()
    {
        cout << "\n Enter name & marks";
        cin >> name >> marks;
    }
    void display()
    {
        this -> get();
        cout << "\n Name =" << this -> name;
        cout << "\n Marks = "<< this -> marks;
    }
};
int main()
{
    test t1, t2;
    t1.display();
    t2.display();
    return 0;
}
```

Output:

```
Enter name & marks
Prajakta 80
Name =Prajakta
Marks = 80
```

6.3.2 Pointer to Objects

[S-18, 22]

- Now we are aware of accessing class members with the help of pointers.
- We can also make a pointer to point to an object created by a class similar to that of normal variables.
- **For example:**

```
class sample
{
    int x, y;
    public:
      void get()
      {
        cin >> x >> y;
      }
```

```
        void put()
        {
          cout << x << y;
        }
    }
```

- Now lets declare a object and a pointer variable of class sample,

```
sample s;
sample * ptr;
```

- Now initialize the pointer ptr with the address of object s.

```
ptr = &s;
```

- With the help of pointer we can access the class members in the same manner as with object.
- Only, the difference is, while accessing class member with the help of objects the (·) dot operator is used where as at a time of pointer (→) arrow operator would be used.
- The following two statements are equivalent.

```
1. s·get()
2. ptr → get();
```

Program 6.13: Write a program to declare class product having data member as product name and product price. Accept and display using pointer to the object.

```
#include<iostream>
#include<conio.h>
class product
{
   protected:
      char pname[40];
      int price;
   public:
      void get()
      {
         cout << "Enter product name and price";
         cin >> pname >> price;
      }
      void display()
      {
         cout << "Product name = \t"<< pname<<endl;
         cout << "Price = \t"<< price;
      }
};
```

```
int main()
{
    product p1, p2, *ptr;
    ptr = &p1;
    p1.get();
    p1.display();
    ptr = &p2;
    p2.get();
    p2.display();
    getch();
    return 0;
}
```

Output:

```
Enter product name and price
Nokia
26500
Product name = Nokia
Price = 26500
```

6.3.3 Pointer to Derived Classes

- We can use pointers not only to the base objects but also to the objects of derived classes pointers to the objects of a base class are type compatible to the objects of 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 derived class from B, then a pointer declared as a pointer to B can also be pointed to D.
- Consider the following declarations:

```
B * cptr;
B b;
D d;
cptr = &b;
```

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

```
cptr = &d;
```

- This is perfectly valid with C++ because `d` is an object derived from the class B. However, there is a problem in using `cptr` to access the public members of the derived class D.
- Using `cptr` we can access only those members inherited from B and not the members that originally belongs to D. In case a member of D has the same name as one of the members of B, any reference to that member by `cptr` will always access the base class members.

- 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 a pointer to derived type.

Program 6.14: Program for pointer to derived class.

```
#include <iostream>
using namespace std;
class BaseClass
 {
    int x;
    public:
       void setx(int i)
    {
       x = i;
    }
    int getx()
    {
       return x;
    }
 };
class DerivedClass: public BaseClass
{
   int y;
    public:
       void sety(int i)
       {
         y = i;
       }
       int gety()
       {
          return y;
       }
};
int main()
{
   BaseClass *p;                    // pointer to BaseClass type
   BaseClass baseObject;            // object of BaseClass
   DerivedClass derivedObject;      // object of DerivedClass
   p = &baseObject;                 // use p to access BaseClass object
   p->setx(10);                     // access BaseClass object
```

```
    cout << "Base object x: " << p->getx();
    p = &derivedObject;              // point to DerivedClass object
    p->setx(99);                     // access DerivedClass object
    derivedObject.sety(88);          // can't use p to set y, so do it directl
    cout << "Derived object x: " << p->getx();
    cout << "Derived object y: " << derivedObject.gety();
    return 0;
}
```

Output:

```
Base object x: 10
Derived object x: 99
Derived object y: 88
Press any key to continue.
```

6.3.4 Virtual Functions [W-18, 22]

- Virtual function is a member function that is declared within a base class and redefined by a derived class. A virtual function is declared by preceding the function declaration in the base class with the keyword **virtual.**

  Example: `virtual void show(); //show function is virtual.`

- When virtual functions access normally, it behaves just like any other type of class member function. However, virtual function is used to support runtime polymorphism when it is accessed via a pointer. A base class pointer can be used to point to an object of any class derived from that base. When different objects are pointed to different versions of the virtual functions are executed. Following program shows the usage of it.

Program 6.15: Program for virtual function.

```
#include<iostream.h>
class base
{
   public:
      virtual void display()
      { cout<<"This is base virtual function \n"; }
};
class derived1: public base
{
   public:
      void display()
      { cout<<"This is derived1 virtual function \n"; }
};
```

```
class derived2: public base
{
   public:
      void display()
      { cout<<"This is derived2 virtual function \n"; }
};
void main()
{
   base * p, b;     // p is base class pointer and b is object of
   derived1 d1;     base
   derived2 d2;
   p = &b;          // point to base
   p → display();  // access base function
   p → & d1;       // point to derived1
   p → display();  // access derived1 function
   p = &d2;         // point to derived2
   p → display();  // access derived2 function
}
```

Output:

```
This is base virtual function
This is derived1 virtual function
This is derived2 virtual function
```

Rules for Virtual Functions:

- The virtual function must be member of same class.
- Virtual functions are accessed by using object pointers.
- They cannot be static members i.e. must be non-static members.
- The prototype of the base class version of a virtual function and all the derived class versions must be identical.
- We can have virtual destructors but do not virtual constructors.
- We cannot use a pointer to a derived class to access object of the base type i.e. reverse is not true.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.
- A virtual function can be a friend of another class.
- Because of the restrictions and differences between function overloading and virtual function redefinition, the term overriding is used to describe virtual functions redefinition by a derived class.

6.3.5 Pure Virtual Functions [S-18, 19, 23]

- Most of the times, the virtual function inside the base class is rarely used for performing any task. It only serves as a placeholder. Such a functions are called do-noting functions; which are pure virtual functions.
- A pure virtual function is a virtual function that has no definition within the base class.
- **Syntax:**

```
virtual <return_type> <function_name> <arg_list> = 0;
```

- When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, compile time error will occur. A class containing such a pure virtual function is called an **Abstract Class.**
- We can define pure virtual function as "The functions which are only declared but not defined in the base class are called as pure virtual function".

Properties of Pure Virtual Functions:

- A pure virtual function has no implementation in the base class, hence a class with pure virtual function cannot be instantiated.
- A pure virtual member function can be invoked by its derived class.
- It is just placeholder for derived class. The derived class is supposed to fill this empty function.

Program 6.16: Program for a pure virtual function.

```
#include <iostream>
using namespace std;
class number
{
    protected:
        int val;
    public:
        void set_val(int i)
        {
            val = i ;

        }
        virtual void show( ) = 0;     // pure virtual function
};
class hextype: public number
{
    public:
        void show( )
        {
        cout<<hex<<val<<"\n";
        }
};
```

```
class dectype: public number
{
    public:
        void show( )
        {
            cout<<val<<"\n";
        }
};
class octtype: public number
{
    public:
        void show()
        {
            cout<<oct<<val<<"\n";
        }
};
int main ()
{
    dectype d;
    hextype h;
    octtype o;
    d.set_val(20);
    d.show();    // displays 20 - decimal
    h.set_val(20);
    h.show();    // displays 14 - hexadecimal
    o.set_val(20);
    o.show();    // displays 24 - octal
    return 0;
}
```

Output:

```
20
14
24
```

- In the above program, as shown the base class, **number** contains an integer called **val**, the function **setval( )** and the pure virtual function **show( )**. The derived classes **hextype**, **dectype** and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (i.e. hexadecimal, decimal or octal).

Summary

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:
  1. **Compile time polymorphism:** This is also called as early or static binding selection of an appropriate function for a particular call at the compile time itself.
  2. **Run-time polymorphism:** This is also called as dynamic binding or late binding. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class.
- Operator overloading refers to overloading of one operator for many different purpose.
- Operator overloading is a compile time polymorphism in which a single operator is overloaded to give user defined meaning to it. Operator overloading provides a flexibility option for creating new definitions of C++ operators.
- Operator overloading is done using operator functions. Operator functions can be a member function or a friend function.
- Operator overloading is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
- Operator overloading mainly comes into picture when the C++ operators are being operated on class objects.
- "The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration. An abstract class is a class in C++ which have at least one pure virtual function.

Check Your Understanding

1. In operator overloading ..... in the basic meaning of operator is not allowed.
   (a) increment (b) addition
   (c) change (d) none
2. If we overload only prefix ++ operator, the postfix ++ operator .....
   (a) does not work (b) works naturally
   (c) works arbitrarily (d) works as if prefix ++ operator

3. When the compiler decides binding as overload member, it is called ..... binding.
   (a) local (b) static
   (c) dynamic (d) safe
4. Following which operator cannot be overloaded?
   (a) + (b) *
   (c) :: (d) both (a) and (b)
5. Operator in C++ can be overloaded by .....
   (a) string function (b) member function
   (c) operator function (d) both (a) & (b)
6. Which of the following is true about this pointer?
   (a) It is passed as a hidden argument to all function calls
   (b) It is passed as a hidden argument to all non-static function calls
   (c) It is passed as a hidden argument to all static functions
   (d) None of the above
7. Polymorphism means .....
   (a) Many forms (b) only one form
   (c) hiding data (d) none of them

Answers

1. (c)	2. (d)	3. (b)	4. (c)	5. (c)	6. (b)	7. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is Operator Overloading?
2. What is meant by 'this' Pointer?
3. Explain rules for operating overloading.
4. What is function overloading?
5. What is Virtual Function?

Q.II Answer the following questions:

1. Why it is necessary to overload an Operator?
2. Explain the rules for Overloading Operators.
3. Explain the usage of this Pointer.
4. What are the C++ operators that can be used for binary and unary applications?
5. Write a C++ program for a class integer which contains an integer as a data member overload the ! operator to find the factorial of an integral.
6. Write short note on: (a) Virtual Function, (b) Pure Virtual Function.
7. Explain this Pointer with example.
8. Write short note on Pointer to Objects.
9. Describe the term Pointer to derived class in detail.
10. What is Pure Virtual Function? Explain with suitable example.
11. What is Polymorphism? What is the difference between Compile Time and Runtime Polymorphism.

Q.III Define the term:

1. Polymorphism
2. Compile time polymorphism
3. Run time polymorphism
4. this pointer
5. Operator overloading

Previous Exam Questions

April 2018

1. Define polymorphism. Explain its types. [4 M]

Ans. Refer to Section 6.1.

2. Explain pointer to object with example. [4 M]

Ans. Refer to Section 6.3.2.

3. Explain pure virtual function with example. [4 M]

Ans. Refer to Section 6.3.5.

4. Define class string using operator overload "= =" to compare two strings. [4 M]

Ans. Refer to Program 6.10.

5. Trace output of the following program and explain it. Assume there is no syntax error: [4 M]

```
#include<iostream.h>
class space
{
    int x, y, z;
    public:
    void getdata(int a, int b, int c)
    {
        x = a;
        y = b;
        z = c;
    }
    void display( )
    {
        cout<< x <<" ";
        cout<< y <<" ";
        cout<< z <<" ";
    }
    void operator - ( )
    {
        x = -x;
        y = -y;
        z = -z;
    }
};
```

```
int main( )
{
    space S;
    S.getdata(10, -20, 30);
    cout<< "S:";
    S.display( );
    -S;
    cout<< "S:";
    S.display( );
    return 0;
}
```

Output: S:10 -20 30 S:-10 20 -30

October 2018

1. What is polymorphism? [2 M]

Ans. Refer to Section 6.1.

2. What is virtual function? [2 M]

Ans. Refer to Section 6.3.4.

3. Write a program to overload unary operator! (NOT). [4 M]

Ans. Refer to Program 6.10.

4. Explain rules for overloading operators. [4 M]

Ans. Refer to Section 6.2.3.

April 2019

1. Define pure virtual function. [2 M]

Ans. Refer to Section 6.3.5.

2. Explain function overloading with example. [4 M]

Ans. Refer to Section 6.2.1.

3. Write a program to overload unary operator. [4 M]

Ans. Refer to Program 6.10.

4. Explain compile time polymorphism and runtime polymorphism with example. [4 M]

Ans. Refer to Section 6.2, 6.3.

5. Create a class time which contains data members as hours, minutes and seconds. Write a C++ program using operator overloading for the following: [4 M]

(i) = = to check whether two times same or not.

(ii) >> to accept time

(iii) << to display time.

Ans. Refer to Program 6.8.