OBJECT ORIENTED PROGRAMMING IN C++

Module 4

Function Overloading

Function overloading is a form of polymorphism. Function overloading facilitates defining one function having many forms. In other words it facilitates defining several functions with the same name, thus overloading the function names. Like in operator overloading, even here, the compiler uses context to determine which definition of an overloaded function is to be invoked.

Function overloading is used to define a set of functions that essentially, do the same thing, but use different argument lists. The argument list of the function is also called as the function's signature. You can overload the function only if their signatures are different.

The differences can be

1. In number of arguments,
2. Data types of the arguments,
3. Order of arguments, if the number and data types of the arguments are same.

e.g.

```
int add( int, int );
int add( int, int, int );
float add( float, float );
float add( int, float, int );
float add(int,int,float);
```

The compiler cannot distinguish if the signature is same and only return type is different. Hence, it is a must, that their signature is different. The following functions therefore raise a compilation error.

e.g.

```
float add( int, float, int );
int add( int, float, int );
```

Consider the following example, which raises a given number to a given power and returns the result.

```
long  long_raise_to(int num, int power)
{
        ng lp = 1;
        for( int  i = 1; i <= power ; i++ )
        {
                lp = lp * num ;
        }
        return lp;
}

double double_raise_to( double num, int power)
{
        double dp = 1;
        for( int  i = 1; i <= power ; i++ )
        {
                dp = dp * num ;
        }
        return dp;
}
```

```
void main(void)
{
        cout<<"2 raise to 8 is <<long_raise_to(2,8);
        cout<<"2.5 raise to 4 is <<double_raise_to(2.5,4);
}
```

In C we would required to write 2 different function names as can be seen from the above example. These functions do identical tasks. However using function overloading techniques we can write the same program in C++ as shown:

```
long  raise_to(int num, int power)
{
        long lp = 1;
        for( int  i = 1; i <= power ; i++ )
        {
                lp = lp * num ;
        }
        return lp;
}


double  raise_to( double num, int power)
{
        double dp = 1;
        for( int  i = 1; i <= power ; i++ )
        {
                dp = dp * num ;
        }
        return dp;
}


void main(void)
{
        cout<<"2 raise to 8 is <<raise_to(2,8);
        cout<<"2.5 raise to 4 is <<raise_to(2.5,4);
}
```

The main () function makes two calls to the function raise_to() –once with 2 integers as arguments and again with an integer and a double. The compiler uses the context to determine which of the functions to invoke. Thus, first time it invokes first function and then the second one.

Precautions with function overloading

Function overloading is a boon to designers, since different names for similar functions need not be thought of, which often is a cumbersome process given that many times people run out of names. But, this facility should not be overused, lest it becomes an overhead in terms of readability and maintenance. Only those functions, which basically do the same task, on different sets of data, should be overloaded.

Call by Reference

Passing variables(parameters) to a function in C can be done in two ways – pass by value, also called as call by value and pass by address or also called as call by address. C++ however , gives one more way to call a function – call by reference. In call by value , a copy of the actual contents of the variables is passed to the function. In this case, changes made to the formal variable , as they are called, are not reflected back in the calling function. Passing an address allows changes to be made directly to the memory, which are occupied by the actual variable.

However, a call by address requires the formal variable to be considered as pointer and thus the indirection operator has to be used with them. A reference on the other hand, is another name, for a previously defined variable.

In other words, after a reference is defined for a particular variable, using its original name as well as the reference can refer to the variable. Hence, a reference is nothing but an alias.

Thus not passing a copy (call by value) saves time and is efficient. It does not have to create a temporary variable /. And not passing the address(call by address), eliminates the use of pointers in the called functions. But, at the same time, the changes made to a formal variable , using its reference or alias can be reflected back in the calling function.

Usage of Reference Variables.

C and C++ , both use the ampersand (&) symbol as the address operator. It is also overloaded to perform binary AND operation on its operands. C++, once more overloads this operator and uses it to declare a reference variable .

e.g.
```
        void main()
        {
                int num1 = 10, num2 = 200;
                int &ref = num1;

                cout << “num1 = “ <<      num1 << endl;
                cout << “num2 = “ <<      num2 << endl;
                cout << “ref = “ <<       ref << endl;  // & used only to declare.
                ref = num2;

                cout << “ After change “ <<endl;
                cout << “num1 = “ <<     num1 << endl;
                cout << “num2 = “ <<     num2 << endl;
                cout << “ref = “  <<     ref  << endl;
        }
```

Output :
```
                num1 = 10
                num2 = 200
                ref = 10

        After change
                num1 = 200
                num2 = 200
                ref = 200
```

A reference variable declared in a function should be initialized at the time of its declaration itself. Also, once a reference has been made to a variable, it cannot be changes to refer to another variable. The program given above shows the result of assigning another variable to a reference variable. the output of the program shows that assigning a variable to a reference variable, actually assigns its contents the original one. Hence , the reference to a particular variable may not be changed later in the program.

Passing Reference to Functions.

Reference variables are particularly useful when passing to functions. The changes made in the called functions are reflected back to the calling function . The program uses the classic problem in programming, swapping the values of two variables.

e.g.

```
        void val_swap(int x, int y)                   // Call by Value
        {
                int t;
                t = x;
```

```
		x = y;
		y = t;
	}

	void add_swap(int *x, int *y)       // Call by Address
	{
		int t;
		t  = *x;
		*x = *y;
		*y = t;
	}

	void val_swap(int &x, int &y)       // Call by Reference
	{
		int t;
		t = x;
		x = y;
		y = t;
	}

	void main()
	{
		int n1 = 25, n2 = 50;
		cout << “Before call by value : “;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;
		val_swap( n1, n2 );
		cout << “ After call by value : “;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;

		cout << “Before call by address : “;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;
		val_swap( &n1, &n2 );
		cout << “ After call by address : “;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;

		cout << “Before call by reference: “;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;
		val_swap( n1, n2 );
		cout << “ After call by value : “ ;
		cout << “ n1 = “ << n1 << “ n2 = “ << n2 << endl;
	}
```

Output :

```
	Before call by value    : n1 = 25  n2 = 50
	After call by value     : n1 = 25  n2 = 50 // x = 50, y = 25

	Before call by address  : n1 = 25  n2 = 50
	After call by address   : n1 = 50  n2 = 25 //x = 50, y = 25

	Before call by reference: n1 = 50  n2 = 25
	After call by reference : n1 = 25  n2 = 50 //x = 25, y = 50
```

You can see that the only difference in writing the functions in call by value and call by reference is while receiving the parameters where as in pass by address the function body has some changes, i.e. they use (*) indirection operator to manipulate the variables.

Returning References from Functions

Just as in passing the parameters by reference, returning a reference also doesn't return back a copy of the variable , instead an alias is returned.

e.g.

```
int &func(int &num)
{
        :
        :
        return(num);
}

void main()
{
        int n1,n2;

        :
        :

        n1 = fn( n2);
}
```

Notice that the function header contains an ampersand (&) before the function name. This is how a function is made to return reference variable. In this case, it takes a reference to an integer as its argument and returns a reference to an integer. This facility can be very useful for returning objects and even structure variables.

Exercises

1. WAP to create a class called as COMPLEX and implement the following by overloading the function ADD which returns a complex number.
   a) ADD(s1, s2) – where s1 is an integer (real) and s2 is a COMPLEX number.
   b) ADD(s1, s2)- where s1 and s2 are COMPLEX numbers.

2. Create a class DateClass with day, month and year as its members. Write an overloaded function for the minus sign. The first function returns the difference between two dates in terms of days.
e.g.

```
DateClass date1, date2;
int diffdays;
diffdays = date1 – date2;
```

The second overloaded function is written which returns a date occurring a given number of days ago.
e.g.

```
DateClass newdate, somedate;
int days;
newdate = somedate – days;
```

Copy Constructor

Similar problems occur at the time of initialization of an object with another of the same class . Although initialization and assignment both assign values, initialization takes place only once when the object is created, whereas assignment can occur whenever desired in the program. Therefore, they differ conceptually.

```
String s1("this is a string");
String s2(s1);                  //Initialization
```

The first statement declares an object s1 and passes a string as an argument to its constructor. The data members of s1 are initialized using the String. The second statement declares another object s2 and contains an object as its argument. Since there is no constructor having an object as its formal argument, the compiler itself initializes the second object's data members with those of the formal object . And since one of the members is a pointer, the problem of assignment arises here too. Overloading the assignment operator solves the problem of assignment. Defining a constructor function that takes object as its argument solves the problem of initialization. This constructor is also called *"copy constructor "*. It is called in three contexts :

- When one object of a class is initialized to another of the same class.
- When an object is passed by value as an argument to a function.
- When a function returns an object.

All these situations result in a copy of one object to another. Hence , a copy constructor is very important if the class contains pointer as data member . A copy constructor for the above example is given below :

```
String (const String &s2)
 {
                              size = s2.size;
                          cptr = new char[size + 1];
                            strcpy(cptr,s2.cptr);
 }
```

It is just like a normal constructor but the argument here is an object of the class . obviously this existing instance should *never* be modified by the copy constructor function , so the keyword *const* should be used in the function signature.

Operator Overloading

All computer languages have built in types like integers, real numbers, characters and so on. Some languages allow us to create our own data types like dates, complex numbers, co-ordinates of a point. Operations like addition, comparisons can be done only on basic data types and not on derived (user-defined) data types. If we want to operate on them we must write functions like compare (), add ().

e.g.

```
if (compare (v1, v2) = = 0)
              :
              :
```

where v1 and v2 are variables of the new data type and compare () is a function that will contain actual comparison instructions for comparing their member variables. However, the concept of Operator Overloading, in C++, allows a statement like

```
  if (v1 = = v2)
              :
              :
```

where the operation of comparing them is defined in a member function and associated with comparison operator(==).

The ability to create new data types, on which direct operations can be performed is called as extensibility and the ability to associate an existing operator with a member function and use it with the objects of its class, as its operands, is called as Operator Overloading.

Operator Overloading is one form of Polymorphism ,an important feature of object-oriented programming .Polymorphism means one thing having many forms, i.e. here an operator can be overloaded to perform different operations on different data types on different contexts. Operator Overloading is also called operational polymorphism. Another form of polymorphism is function overloading.

Operator Overloading Fundamentals

The C language uses the concept of Operator Overloading discreetly. The asterisk (*) is used as multiplication operator as well as indirection (pointer) operator. The ampersand (&) is used as address operator and also as the bitwise logical ‘AND’ operator. The compiler decides what operation is to be performed by the context in which the operator is used.

Thus, the C language has been using Operator Overloading internally. Now, C++ has made this facility public. C++ can overload existing operators with some other operations. If the operator is not used in the context as defined by the language, then the overloaded operation, if defined will be carried out.
For example, in the statement

```
x = y + z;
```

If x, y and z are integer variables, then the compiler knows the operation to be performed. But, if they are objects of some class, then the compiler will carry out the instructions, which will be written for that operator in the class.

Implementing Operator Functions

The general format of the Operator function is:

return_type operator **op** (argument list);

Where *op* is the symbol for the operator being overloaded. Op has to be a valid C++ operator, a new symbol cannot be used.

e.g.

Let us consider an example where we overload unary arithmetic operator ‘++’.

```
class Counter
{
        public :
                Counter();
                void operator++(void);
        private :
                int Count;

};


Counter::Counter()
{
        Count = 0;
}

void Counter::operator++(void)
{
        ++ Count ;
}

void main()
{
        Counter c1;
        c1++;                   // increments Count to 1
        ++c1;                   // increments Count to 2
 }
```

In main() the increment operator is applied to a specific object. The function itself does not take any arguments. It increments the data member Count. Similarly, to decrement the Counter object can also be coded in the class definition as:

```
void operator--(void)
 {
        -- Count ;
 }
```

and invoked with the statement

```
--c1; or c1--;
```

In the above example , the compiler checks if the operator is overloaded and if an operator function is found in the class description of the object, then the statement to increment gets converted, by the compiler, to the following:

```
c1.operator++();
```

This is just like a normal function call qualified by the object's name. It has some special characters (++) in it. Once this conversion takes place, the compiler treats it just like any other member function from the class. Hence, it can be seen that such a facility is not a very big overhead on the compiler.

However, the operator function in the above example has a potential glitch. On overloading , it does not work exactly like it does for the basic data types. With the increment and decrement operators overloaded, the operator function is executed first, regardless of whether the operator is postfix or prefix.

If we want to assign values to another object in main() we have to return values to the calling function.

```
Counter Counter :: operator++(void)
{
        Counter temp;
        temp.Count = ++ Count ;
        return ( temp );
}

void main()
{
        Counter c1,c2;
        c1 = c2 ++;       //increments to 1, then assigns.
}
```

In this example , the operator function creates a new object *temp* of the class Counter, assigns the incremented value of Count to the data member of temp and returns the new object. This object is returned to main(). We can do this in another way by creating a nameless temporary object and return it.

```
class Counter
{
        public :
                Counter();   // CONSTRUCTOR WITHOUT ARGUMENTS
                Counter( int c); // CONSTRUCTOR WITH 1 ARGUMENT
                Counter operator++(void);

        private :
                int Count;
};

Counter::Counter()   // CONSTRUCTOR WITHOUT ARGUMENTS
{
        Count = 0;
}

Counter::Counter( int c) // CONSTRUCTOR WITH 1 ARGUMENT
{
        Count = c;
}

Counter Counter::operator++(void)
{
        ++ Count ;
        return Counter(Count);
}
```

One change we can see is a constructor with one argument. No new temporary object is explicitly created. However return statement creates an unnamed temporary object of the class Counter initializes it with the value in Count and returns the newly created object. Hence one argument constructor is required.

Yet another way of returning an object from the member function is by using the *this* pointer. This special pointer points to the object, which invokes the function. The constructor with one argument is not required in this approach.

```
Counter Counter :: operator++(void)
{
        ++ Count ;
        return ( * this);
}
```

Consider a class COMPLEX for Complex numbers. It will have a real and an imaginary member variable. Here we can see *binary operator overloaded and also how to return values from the functions.*

```
class  COMPLEX
{
        public:
                COMPLEX operator+(COMPLEX);

        private:
                int real, imaginary;
}
```

Suppose that C1, C2 and C3 are objects of this class. Symbolically addition can be carried out as

```
C3 = C1 + C2;
```

The actual instructions of the operator are written in a special member function.

e.g.

```
COMPLEX COMPLEX :: operator+( COMPLEX C2)
{
        COMPLEX temp;
        temp.real      = real + C2.real;
        temp.imaginary = imaginary + C2. imaginary;
        return (temp);
}
```

The above example shows how Operator Overloading is implemented. It overloads “+” operator to perform addition on objects of COMPLEX class. Here we have overloaded a binary operator(+).

Rules for overloading an operator

This summarizes the most important points you need to know in order to do operator function overloading.

- The only operators you may overload are the ones from the C++ list and not all of those are available. You cannot arbitrarily choose a new symbol (such as @) and attempt to “overload it.
- Start by declaring a function in the normal function fashion, but for the function name use the expression:

  Operator *op*

  Where *op* is the operator to be overloaded. You may leave one or more spaces before *op*.
- The pre-defined precedence rules cannot be changed. i.e. you cannot, for example, make binary ‘+’ have higher precedence than binary ‘*’. In addition, you cannot change the associativity of the operators.
- The unary operators that you may overload are:

->	indirect member operator
!	not
&	address
*	dereference
+	plus
-	minus
++	prefix increment
++	postfix increment (possible in AT & T version 2.1)
--	postfix decrement
--	prefix decrement (possible in AT & T version 2.1)
~	one’s complement

- The binary operators that you may overload are:
  (), [], new, delete, *, / , %, + , - , <<,>>, <, <=, >, >=, ==,! =, &, ^, |, &&, ||, =, *=, /=, %=, +=, -, =, <<=, >>=, &=,! =, ^=, ','(Comma).
- The operators that can not be overloaded are:

.	direct member
.*	direct pointer to member
::	scope resolution
?:	ternary

- No default arguments are allowed in overloaded operator functions.
- As with the predefined operators, an overloaded operator may be unary or binary. If it is normally unary, then it cannot be defined to be binary and vice versa. However, if an operator can be both unary and binary, then it can be overloaded either way or both.
- The operator function for a class may be either a non-static member or global friend function. A non-static member function automatically has one argument implicitly defined, namely the address of the invoking instance (as specified by the pointer variable *this***)**. Since a friend function has no *this* pointer, it needs to have all its arguments explicitly defined).
- At least one of the arguments to the overloaded function explicit or implicit must be an instance of the class to which the operator belongs.

Consider an example, which depicts overloading of += (Compound assignment), <, >, == (Equality),!=, + (Concatenation) using String class.

```
class String
{
public :
            String ();
            String ( char str [] );
            void putstr();
            String operator + (String);
            String operator += (String s2);
            int operator < (String s2);
            int operator > (String s2);
            int operator == (String s2);
            int operator != (String s2);

private :
            char s[100];
};


String::String () // CONSTRUCTOR WITH NO ARGUMENTS
{
            s[0] = 0;
};


String:: String( char str [] ) // CONSTRUCTOR WITH ONE
ARGUMENT
{
            strcpy(s,str)
};

void String:: putstr()           // FUNCTION TO PRINT STRING
{
            cout << s ;
};

String String :: operator+(String s2)
{
            String temp;
            strcpy(temp.s,s);
            strcat(temp.s,s2.s);
            return (temp);
}

String String :: operator+=(String s2)
{

            strcat(s,s2.s);
            return (*this);
}

int String::operator < (String s2)
{
            return (strcmp (s, s2.s ) < 0);
}

int String::operator > (String s2)
{
            return (strcmp (s, s2.s ) > 0);
}

int String::operator == (String s2)
{
            return (strcmp (s, s2.s ) == 0);
}

int String::operator != (String s2)
{
            return (strcmp (s, s2.s ) != 0);
}

void main()
{
            String s1 = “welcome “;
            String s2 = “ to the world of c++”;
            String s3;

            cout << endl << “s1 = “;
            s1.putstr();
            cout << endl << “s2 = “;
            s2.putstr();

            s3 = s1 + s2;
            cout << endl << “ s3 = “;
            s3.putstr();

            String s4;

            cout <<endl<<” *********************”;
            s4 = s1 + = s2;

            cout << endl << “ s4 = “;
            s4.putstr();

            String s5 = “ Azzzz “;
            String s6 = “ Apple “;

            if( s5 < s6 )
            {
                        s5.putstr();
                        cout << ” < ”;
                        s6.putstr();
            }
            else if( s5 > s6 )
            {
                        s5.putstr();
                        cout << ” > ”;
                        s6.putstr();
            }
            else if( s5 == s6 )
            {
                        s5.putstr();
                        cout << ” = ”;
                        s6.putstr();
            }
            else if( s5 != s6 )
            {
                        s5.putstr();
                        cout << ” < ”;
                        s6.putstr();
            }
}
```

Output:

```
            S1 = welcome
            S2 = to the world of C++
            S3 = welcome to the world of c++
            *************************
            S4 = welcome to the world of c++
```

Operator Overloading Examples

1. Program for Overloading **" + operator"**.

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator + (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator + (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {

        temp.element[i]=element[i]+a.element[i];
        }
        return (temp);
}


void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x+y;
        cout<<"\n\n Array Addition :";
        z.putdata();
}
```

2. Program for overloading " **- operator**".

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator - (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator - (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {
                temp.element[i]=element[i]-
a.element[i];
        }
        return (temp);
}

void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x-y;
        cout<<"\n\n Array Subtraction :";
        z.putdata();
}
```

3. Program for overloading “ *** operator**”.

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator * (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator * (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {

        temp.element[i]=element[i]*a.element[i];
        }
        return (temp);
}


void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x*y;
        cout<<"\n\n Array Multiplication :";
        z.putdata();
}
```

4. Program for overloading “ **/ Operator**”.

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator / (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator / (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {

        temp.element[i]=element[i]/a.element[i];
        }
        return (temp);
}

void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x/y;
        cout<<"\n\n Array Division :";
        z.putdata();
}
```

5. Program for overloading “**% operator**”

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator % (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator % (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {

        temp.element[i]=element[i]%a.element[i]
;
        }
        return (temp);
}

void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x%y;
        cout<<"\n\n Array Modulo Division :";
        z.putdata();
}
```

6. Program for overloading “ **^ Operator**”.

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

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                array operator ^ (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

array array::operator ^ (array a)
{
        array temp;
        for(int i=0;i<5;i++)
        {

        temp.element[i]=pow(element[i],a.eleme
nt[i]);
        }
        return (temp);
}

void main()
{
        array x,y,z;
        x.getdata();
        y.getdata();
        z=x^y;
        cout<<"\n\n Array Power :";
        z.putdata();
}
```

7. Program for overloading “ **++ operator**”.

```
# include <iostream.h>

class item
{
        private:
                int mem1,mem2;
        public:
                item(int x,int y);
                void operator ++();     //To
Support Prefix Operation
                void operator ++(int z);//To
Support Postfix Operation
                void display();
};

item::item(int x, int y)
{
        mem1=x;
        mem2=y;
}

void item::operator ++ ()
{
        ++mem1;
        ++mem2;
}

void item::operator ++(int z)
{
        mem1++;
        mem2++;
}

void item::display()
{
        cout<<"\n Member 1 :"<<mem1;
        cout<<"\n Member 2 :"<<mem2;
}


void main()
{
        item a(10,50);
        cout<<"\n Before Overloading";
        a.display();
        ++a;
        cout<<"\n After Overloading Prefix Form";
        a.display();
        a++;
        cout<<"\n After Overloading Postfix Form";
        a.display();

}
```

8. Program for overloading “ **+= Operator**”

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                void operator += (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

void array::operator += (array a)
{
        for(int i=0;i<5;i++)
        {
                element[i]+=a.element[i];
        }
}

void main()
{
        array x,y;
        x.getdata();
        y.getdata();
        x+=y;
        cout<<"\n\n Array Addition : += ";
        x.putdata();
}
```

9. Program for overloading “ **-= Operator**”.

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                void operator -= (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

void array::operator -= (array a)
{
        for(int i=0;i<5;i++)
        {
                element[i]-=a.element[i];
        }
}

void main()
{
        array x,y;
        x.getdata();
        y.getdata();
        x-=y;
        cout<<"\n\n Array Subtraction : -= ";
        x.putdata();
}
```

10. Program for overloading “ ***= Operator**”

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                void operator *= (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

void array::operator *= (array a)
{
        for(int i=0;i<5;i++)
        {
                element[i]*=a.element[i];
        }
}

void main()
{
        array x,y;
        x.getdata();
        y.getdata();
        x*=y;
        cout<<"\n\n Array Multiplication : *= ";
        x.putdata();
}
```

11. Program for overloading “ **/= Operator**”.

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                void putdata();
                void operator /= (array);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

void array::putdata()
{
        for(int i=0;i<5;i++)
        {
                cout<<"\t"<<element[i];
        }
}

void array::operator /= (array a)
{
        for(int i=0;i<5;i++)
        {
                element[i]/=a.element[i];
        }
}

void main()
{
        array x,y;
        x.getdata();
        y.getdata();
        x/=y;
        cout<<"\n\n Array Division : /= ";
        x.putdata();
}
```

12. Program for overloading “ **[] Operator**”

```
# include <iostream.h>

class array
{
        private:
                int element[5];
        public:
                array();
                void getdata();
                int operator [] (int x);
};

array::array()
{
        for(int i=0;i<5;i++)
        {
                element[i]=0;
        }
}

void array::getdata()
{
        cout<<"\n Enter Elements :";
        for(int i=0;i<5;i++)
        {
                cin>>element[i];
        }
}

int array::operator [] (int x)
{
        return (element[x]);
}

void main()
{
        array example;
        example.getdata();
        for(int i=0;i<5;i++)
        {
                cout<<"\n Location "<<i+1<<"
Value : "<<example[i];
        }
}
```

13. Program for overloading “ **== Operator** ”

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

class strings
{
        private:
                char str[80];
        public:
                strings();
                void getdata();
                int operator ==(strings a);
};

strings::strings()
{
        strcpy(str,"");
}

void strings::getdata()
{
        cin>>str;
}

int strings::operator ==(strings a)
{
        if((strcmp(str,a.str)==0))
        {
                return 1;
        }
        else
        {
                return 0;
        }
}

void main()
{
        strings x,y;
        cout<<"\n\n Enter First String :";
        x.getdata();
        cout<<"\n\n Enter Second String :";
        y.getdata();
        if(x==y)
        {
                cout<<"\n\n Strings are Equal";
        }
        else
        {
                cout<<"\n\nStrings are not Equal";
        }
}
```

14. Program for overloading “ **>= Operator** ”

```
# include <iostream.h>

class item
{
        private:
                int element;
        public:
                item(int val);
                int operator >=(item a);
};

item::item(int val)
{
        element=val;
}

int item::operator >=(item a)
{
        if(element>=a.element)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}

void main()
{
        item x(50),y(30);
        if(x>=y)
        {
                cout<<"X is greater";
        }
        else
        {
                cout<<"Y is greater";
        }
}
```

15. Program for overloading “ **<= Operator**”.

```
# include <iostream.h>

class item
{
        private:
                int element;
        public:
                item(int val);
                int operator <=(item a);
};

item::item(int val)
{
        element=val;
}

int item::operator <=(item a)
{
        if(element<=a.element)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}

void main()
{
        item x(50),y(30);
        if(x<=y)
        {
                cout<<"X is Smaller";
        }
        else
        {
                cout<<"Y is Smaller";
        }
}
```

16. Program for overloading “ **!= Operator** “

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

class item
{
        private:
                char str[80];
        public:
                item(char *s);
                int operator != (item x);
};

item::item(char *s)
{
        strcpy(str,s);
}

int item::operator != (item x)
{
        if(strcmp(str,x.str)==0)
        {
                return 0;
        }
        else
        {
                return 1;
        }
}

void main()
{
        item a("Hi"),b("Hiii");
        if(a!=b)
        {
                cout<<"\n Strings are Not Equal";
        }
        else
        {
                cout<<"\n Strings are Equal";
        }
}
```

17. Program for overloading " **() operator**"

```
# include <iostream.h>
{
        private:
                int val1, val2;
        public:
                item(int x, int y);
                int operator()();
};

item::item(int x, int y)
{
        val1=x;
        val2=y;
}

int item::operator()()
{
        return((val1+val2)/2);
}

void main()
{
        item average(50,80);
        cout<<"\n Average = "<<average();
}
```

18. Program for overloading casting operator eg: **(int)**

```
# include <iostream.h>

class item
{
        private:
                int val1, val2;

        public:
                item(int x, int y);
                operator int();
};

item::item(int x, int y)
{
        val1=x;
        val2=y;
}

item::operator int ()
{
        return(val1+val2);
}

void main()
{
        item a(30,10);
        int c=(int)(a);
        cout<<"\n Value after Casting :"<<c;
}
```

19. Program for overloading "**Unary Operator – "**

```
# include <iostream.h>

class item
{
        private:
                int element1,element2;
        public:
                item(int val1, int val2);
                void operator -();
};

item::item(int val1, int val2)
{
        element1=val1;
        element2=val2;
}

void item::operator -()
{
        element1=-1*element1;
        element2=-1*element2;
}

void main()
{
        item x(50,30);
        -x;
        x.putdata();
}
```

20. Program for overloading “ **<< Operator “**

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

class item
{
        private:
                char str[80];
        public:
                item(char *s);
                void display();
                void operator << (item x);
};

item::item(char *s)
{
        strcpy(str,s);
}

void item::display()
{
        cout<<"\n\nString : "<<str;
}

void item::operator << (item x)
{
        strcpy(str,x.str);
}

void main()
{
        item text1("Rajagiri School"),text2("");
        text2<<text1;
        text2.display();
}
```

21. Program for overloading “ **>> Operator** “

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

class item
{
        private:
                char str[80];
        public:
                item(char *s);
                void display();
                void operator >> (item *x);
};

item::item(char *s)
{
        strcpy(str,s);
}

void item::display()
{
        cout<<"\n\nString : "<<str;
}

void item::operator >> (item *x)
{
        strcpy(x->str,str);
}

void main()
{
        item text1("Rajagiri School"),text2("");
        text1>>&text2;
        text2.display();
}
```

Friend Functions

One of the main features of OOP is information hiding. A class encapsulates data and methods to operate on that data in a single unit . The data from the class can be accessed only through member functions of the class. This restricted access not only hides the implementation details of the methods and the data structure, it also saves the data from any possible misuse, accidental or otherwise. However, the concept of data encapsulation sometimes takes information hiding too far. There are situations where a rigid and controlled access leads to inconvenience and hardships.

For instance, consider a case where a function is required to operate on object of two different classes. This function cannot be made a member of both the classes. What can be done is that a function can be defined outside both the classes and made to operate on both. That is, a function not belonging to either, but able to access both. Such a function is called as a friend function. In other words, a friend function is a nonmember function, which has access to a class’s private members. It is like allowing access to one’s personal belongings to a friend.

Using a Friend

Using a friend function is quite simple. The following example defines a friend function to access members of two classes.

```
class Bclass;                           // Forward Declaration

class Aclass
{
        public :
                Aclass(int v)
                {
                        Avar = v;
                }

                friend int addup(Aclass &ac, Bclass &bc);
        private :
                int Avar;
};

class Bclass
{
        public :
                Bclass(int v)
                {
                        Bvar = v;
                }

                friend int addup(Aclass &ac, Bclass &bc);
        private :
                int Bvar;

};


int addup(Aclass &ac, Bclass &bc)
{
        return( ac.Avar + bc.Bvar);
}

void main()
{
        Aclass aobj;
        Bclass bobj;

        int total;
        total = addup(aobj,bobj);
}
```

The program defines two classes- Aclass and Bclass. It also has constructors for these classes. A friend function, addup(), is declared in the definition of both the classes, although it does not belong to either. This friend function returns the sum of the two private members of the classes. Notice, that this function can directly access the private members of the classes. To access the private members, the name of the member has to be prefixed with the name of the object , along with the dot operator.

The first line in the program is called forward declaration. This is required to inform the compiler that the definition for class Bclass comes after class Aclass and therefore it will not show any error on encountering the object of Bclass in the declaration of the friend function. Since it does not belong to both the classes , while defining it outside we do not give any scope resolution and we do not invoke it with the help of any object. Also , the keyword friend is just written before the declaration and not used while defining.

Sometimes friend functions can be avoided using inheritance and they are preferred. Excessive use of friend over many classes suggests a poorly designed program structure. But sometimes they are unavoidable.

Friend for Overloading Operators

Some times friend functions cannot be avoided. For instance with the operator overloading. Consider the following class that contains data members to simulate a matrix. Several operations can be performed on the matrices. One of them is to multiply the given matrix by a number(constant literal). There are two ways in which we can do this. The two ways are :

```
Matrix * num;

  [or]

num * Matrix;
```

In the first case we can overload * to perform the operation and an object invokes this as the statement gets converted to :

```
Mobj.operator*(num);
```

Where Mobj is an object of Matrix and num is a normal integer variable. What happens to the second one ? It gets converted by the compiler as :

```
num.operator*(Mobj);
```

and compiler will give an error message, because num is not an object. To support above statement we are using friend function in operator overloading.

Let us see this program in detail.

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

// This is a program to add a scalar value to an object
// This program uses both function overloading and operator overloading
// Operator overloading using friend functions

class item
{
        private:
                int x;
        public:
                item(int var)
                {
                        x=var;
                }
                friend int operator + (int par,item Y);
                friend int operator + (item Y,int par);
};

// Function Definition to support 2+X
int operator + (int par,item Y)
{
        return(Y.x+par);
}

// Function Definition to support X+2
int operator + (item Y, int par)
{
        return(Y.x+par);
}

void main()
{
        clrscr();
        item X(5);
        cout<<X+2;//Friend function call -> operator +(X,2);
        cout<<2+X;//Friend function call -> operator +(2,X);
        getch();
}
```

Here when the compiler comes across the addition of an object by a number it invokes the friend function.

Granting friendship to another class

To grant friendship to another class, write the keyword followed by the class name. The keyword class is optional. Note that this declaration also implies a forward declaration of the class to which the friendship is being granted. The implication of this declaration is that all off the member functions of the friend class are friend functions of the class that bestows the friendship.

e.g.

```
class Aclass
{
        public :
                        :
                        :
                        :

                friend class Bclass;   // Friend declaration.
        private :
                int Avar;

};

class Bclass
{
        public :
                        :
                        :
                void fn1(Aclass ac)
                {
                        Bvar = ac. Avar;   // Avar can be accessed.
                }

        private :
                int Bvar;

};

void main()
{
        Aclass aobj;
        Bclass bobj;
        Bobj,fn1(aobj);
}
```

The program declares class Bclass to be a friend of Aclass. It means that all member functions of Bclass have been granted direct access to all member functions of Aclass.

Granting friendship to a member function of another class

If you want class A to grant friendship to one or more individual member functions of class B, then you must code the classes and their member functions in this manner:

- Forward declare class A;
- Define class B and declare (not define) the member functions:
- Define class A in which you declare the friendship for the member functions of class B. Of course, you must qualify the names of these functions using the class name B and the scope resolution operator.
- Define the member functions of class B;

e.g.

```
class Aclass
class Bclass
{
        public :
                        :
                        :

                void fn1();     // Can't define here
                void fn3()
                {
                        :
                        :
                }

        private :
                int Bvar;
};

class Aclass
{
        public :
                        :
                        :


                void fn1()
                {
                        :
                }
                friend classB:: fn1();
                friend classB:: fn2();

        private :
                int Avar;
};

void classB:: fn1()
{
        Bvar = Avar;
}

void classB:: fn2()
{
        Bvar = variable +25;
}
```

Conversion Functions

Conversion functions are member functions used for the following purposes:

1. Conversion of object to basic data type.
2. Conversion of basic data type to object.
3. Conversion of objects of different classes.

Conversions of one basic data type to another are automatically done by the compiler using its own built-in routines (implicit) and it can be forced using built-in conversion routines (explicit). However, since the compiler does not know anything about user-defined types (classes), the program has to define conversion functions.

e.g.

```
int i = 2, j =19;
float f = 3.5;

i = f;  // i gets the value 3 , implicit conversion
f = (float) j; // f gets 19.00, explicit conversion
```

Conversion from Basic to User-Defined variable

Consider the following example.

```
class Distance
{
        public  :
                Distance(void)    // Constructor with no argument
                {
                        feet = 0;
                        inches = 0.0;
                 }

                Distance(float metres)
                {
                        float f;                        // Constructor with one argument also used for conversion
                        f = 3.28 * metres;
                        feet = int(f);
                        inches = 12 * ( f – feet);
                }

                void display(void)
                {
                        cout << " Feet = " << feet <<",";
                        cout << " Inches = " << inches << endl;
                 }

        private :
                int feet;
                float inches;
};


void main (void)
{
        Distance d1 = 1.25;       // Uses 2nd constructor
        Distance d2;              // Uses 1st constructor
        float m;
```

```
            d2 = 2.0 ;                    // Uses 2nd constructor

            cout << " 1.25 metres is : " << d1.showdist() ;
            cout << " 2.0  metres is :" << d2.showdist();
    }
```

Output :

```
            1.25 metres is :FEET = 4 , INCHES = 1.199999
            2.0  metres is :FEET = 6 , INCHES = 6.719999
```

The above program converts distance in metres (basic data type) into feet and inches (members of an object of class Distance).

The declaration of first object d1 uses the second constructor and conversion takes place. However, when the statement encountered is

```
            d2 = 2.0;
```

The compiler first checks for an operator function for the assignment operator. If the assignment operator is not overloaded, then it uses the constructor to do the conversion.

Conversion from User-Defined to Basic data type

The following program uses the program in the previous section to convert the Distance into metres(float).

```
    class Distance
    {
            public  :
                    Distance(void)    // Constructor with no argument
                    {
                            feet = 0;
                            inches = 0.0;
                    };

                    Distance(float metres)
                    {
                            float f;              // Constructor with one argument Also used for conversion
                            f = 3.28 * metres;
                            feet = int(f);
                            inches = 12 * ( f – feet);
                    }

                    operator float(void)        // Conversion function from Distance to float
                    {
                            float f;
                            f = inches / 12;
                            f = f + float (feet);
                            return ( f/ 3.28 );
                    }

                    void display(void)
                    {
                            cout << " Feet = " << feet <<",";
                            cout << " Inches = " << inches << endl;
                    }
    private :
```

```
                int feet;
                float inches;

        };



        void main (void)
        {
                Distance d1 = 1.25;                 // Uses 2nd constructor
                Distance d2;                        // Uses 1st constructor
                float m;

                d2 = 2.0 ;                          // Uses 2nd constructor

                cout << " 1.25 metres is :" << d1.showdist ();
                cout << " 2.0  metres is :" << d2.showdist ();

                cout << " CONVERTED BACK TO METRES ";

                m = float ( d1 );                   // Calls function explicitly.
                cout << " d1 = " << m;

                m = d2;                             // Calls function explicitly.
                cout << " d2 = " << m;
        }
```

Output:

```
        1.25 metres is :FEET = 4 ,INCHES = 1.199999
        2.0  metres is :FEET = 6 ,INCHES = 6.719999

        CONVERTED BACK TO METRES

        d1 = 1.25
        d2 = 2.00
```

Actually, this conversion function is nothing but overloading the typecast operator float(). The conversion is achieved explicitly and implicitly.

```
        m = float (d1);
```

is forced where as , in the second assignment statement

```
        m = d2;
```

first the compiler checks for an operator function for assignment (=) operator and if not found it uses the conversion function.

The conversion function must not define a return type nor should it have any arguments.

Inheritance

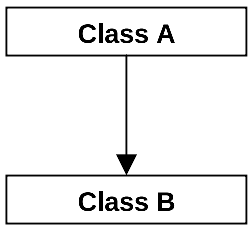
Object-oriented programming as seen in the preceding sessions emphasizes the data, rather than emphasizing algorithms. The previous sessions covered OOP features like extensibility, data encapsulation, information hiding, functional polymorphism and operational polymorphism. OOP, however, has more jargon associated to it, like reusability, inheritance. This session covers reusability and inheritance.

Reusability

Reusability means reusing code written earlier ,may be from some previous project or from the library. Reusing old code not only saves development time, but also saves the testing and debugging time. It is better to use existing code, which has been time-tested rather than reinvent it. Moreover, writing new code may introduce bugs in the program. Code, written and tested earlier, may relieve a programmer of the nitty-gritty. Details about the hardware, user-interface, files and so on. It leaves the programmer more time to concentrate on the overall logistics of the program.

What is inheritance?

Class, the vehicle, which is used to implement object-oriented concepts in C++, has given a new dimension to this idea of reusability. Many vendors now offer libraries of classes. A class library consists of data and methods to operate on that data, encapsulated in a class . The source code of these libraries need not be available to modify them. The new dimension of OOP uses a method called inheritance to modify a class to suit one's needs. Inheritance means deriving new classes from the old ones. The old class is called the *base class* or *parent class or super class* and the class which is derived from this base class is called as *derived class* or *child class or sub class.* Deriving a new class from an existing one , allows redefining a member function of the base class and also adding new members to the derived class . and this is possible without having the source of the course definition also. In other words, a derived class not only inherits all properties of the base class but also has some refinements of its own. The base class remains unchanged in the process. In other words, the derived class "is a " type of base class, but with more details added. For this reason, the relationship between a derived class and its base class is called an "is-a" relationship.



Single Inheritance

Here class A is a base class and the class B is the derived class.

How to define a derived class ?

A singly inherited derived class id defined by writing :

- The keyword *class.*
- The name of the derived class .
- A single colon (:).
- The type of derivation (private , protected, or public).
- The name of the base, or parent class.
- The remainder of the class definition.

e.g.

```
class A
{
        public :
                int public_A;
                void public_function_A();
        private :
                int pri_A;
                void private_function_A();

        protected :
                int protected_A;
                void protected_function_A();

 };

class B : private  A
{
        public :
                int public_B;
                void public_function_B();

        private :
                int pri_B;
                void private_function_B();
};

class C : public  A
{
        public :
                int public_C;
                void public_function_C();

        private :
                int pri_C;
                void private_function_C();

};

class D : protected A
{
        public :
                int public_D;
                void public_function_D();
        private :
                int pri_D;
                void private_function_D();

};
```

A derived class always contains all of the member members from its base class . you cannot “subtract” anything from a base class. However, accessing the inherited variables is a different matter. It is also important to understand the privileges that the derived class has insofar as access to members of its base class are concerned. In other words, just because you happen to derive a class does not mean that you are automatically granted complete and unlimited access privileges to the members of the base class. to understand this you must look at the different types of derivation and the effect of each one.

Private derivation

If no specific derivation is listed, then a private derivation is assumed. If a new class is derived privately from its parent class , then :

- The private members inherited from its base class are inaccessible to new member functions in the derived class . this means that the creator of the base class has absolute control over the accessibility of these members , and there is no way that you can override this.

- The public members inherited from the base class have private access privilege. In other words, they are treated as though they were declared as new private members of the derived class, so that new member functions can access them. However, if another private derivation occurs from this derived class, then these members are inaccessible to new member functions.

e.g.

```
class base
{
        private :
                int number;
};

class derived : private base
{
        public :
                void f()
                {
                        ++number;    // Private base member not accessible

                }
};
```

The compiler *error* message is

‘ base :: number ‘ is not accessible in the function derived :: f();

e.g.

```
class base
{
  public :
                int number;
};

class derived : private base
{
        public :
                void f()
                {
                        ++number;    // Access to number O.K.
                }
};

class derived2 : private derived
{
        public :
                void g()
                {
                        ++number;    // Access to number is prohibited.

                }
};
```

The compiler *error* message is

‘ base :: number ‘ is not accessible in the function derived2 :: g();

Since public members of a base class are inherited as private in the derived class, the function derived :: f() has no problem accessing it . however, when another class is derived from the class derived , this new class inherits number but cannot access it. Of course, if derived1::g() were to call upon derived::f(), there is no problem since derived::f() is public and inherited into derived2 as private.

i.e. In derived2 we can write,

```
void g()
{
        f();
}
```

or there is another way. Writing access declaration does this.

```
class base
{
        public :
                int number;
};

class derived : private base
{
        public : base :: number ;
        void f()
        {
                ++number;     // Access to number O.K.
        }
 };

class derived2 : private derived
{
        public :
                void g()
                {
                        ++number;     // Access to number O.K
                }
};
```

As you have just seen private derivations are very restrictive in terms of accessibility of the base class members . therefore, this type of derivation is rarely used.

Public derivation

Public derivations are much more common than private derivations. In this situation :

- The private members inherited from the base class are inaccessible to new members functions in the derived class.
- The public members inherited from the base class may be accessed by new members functions in the derived class and by instances of the derived class .

e.g.

```
class base
{
        private :
                int number;
};

class derived : public base
{
        public :

                void f()
                {
                        ++number;     // Private base member not accessible
                }
};
```

The compiller *error* message is

‘ base :: number ‘ is not accessible in the function derived::f();

Here, only if the number is public then you can access it.

Note : *However example 2 and 3 in the above section works here if you derive them as “public”.*

The Protected Access rights

In the preceding example, declaring the data member *number* as private is much too restrictive because clearly new members function in the derived class need to gain access to it and in order to perform some useful work.

To solve this dilemma, the C++ syntax provides another class access specification called *protected .* here is how protected works :

- In a private derivation the protected members inherited from the base class have private access privileges. Therefore, new member functions and friend of the derived class may access them.
- In a public derivation the protected members inherited from the base class retain their protected status. They may be accessed by new members function and friends of the derived class .

In both situations the new members functions and friends of the derived class have unrestricted access to protected members . however, as the instances of the derived class are concerned, protected and private are one and the same, so that direct access id always denied. Thus, you can see that the new category of protected provides a middle ground between public and private by granting access to new function and friends of the derived class while still blocking out access to non-derived class members and
friend functions .

```
class base
{
        protected :
                int number;
};

class derived : public base
{
        public :
                void f()
                {
                        ++number;     // base member access O.K.
                }
};
```

Protected derivation

In addition to doing private and public derivations, you may also do a protected derivation. In this situation :

- The private members inherited from the base class are inaccessible to new member functions in the derived class.
  (this is exactly same as if a private or public derivation has occurred.)
- The protected members inherited from the base class have protected access privilege.
- The public members inherited from the base class have protected have protected access.

Thus , the only difference between a public and a protected derivation is how the public members of the parent class are inherited. It is unlikely that you will ever have occasion to do this type of derivation.

Summary of access privileges

1. If the designer of the base class wants no one, not even a derived class to access a member , then that member should be made private .
2. If the designer wants any derived class function to have access to it, then that member must be protected.
3. if the designer wants to let everyone , including the instances, have access to that member , then that member should be made public .

Summary of derivations

1. Regardless of the type of derivation, private members are inherited by the derived class , but cannot be accessed by the new member function of the derived class , and certainly not by the instances of the derived class .
2. In a private derivation, the derived class inherits public and protected members as private . a new members function can access these members, but instances of the derived class may not. Also any members of subsequently derived classes may not gain access to these members because of the first rule.
3. In public derivation, the derived class inherits public members as public , and protected as protected . a new member function of the derived class may access the public and protected members of the base class ,but instances of the derived class may access only the public members.
4. In a protected derivation, the derived class inherits public and protected members as protected .a new members function of the derived class may access the public and protected members of the base class, both instances of the derived class may access only the public members .

Table of Derivation and access specifiers

Derivation Type	**Base Class Member**	**Access in Derived Class**
Private	Private	(inaccessible)
	Public	Private
	Protected	Private
Public	Private	(inaccessible)
	Public	Public
	Protected	Protected
Protected	Private	(inaccessible)
	Public	Protected
	Protected	Protected

Using the Derived Class

An instance of a derived class has complete access to the public members of the base class . assuming that the same name does not exist within the scope of the derived class , the members from the base class will automatically be used. Because there is no ambiguity involved in this situation, you do not need to use scope resolution operator to refer to this base class member.

```
class base
{
        public :
                base(int n = 0)
                {
                        number = n;
                }
                int get_number();

        protected :
                int number;

};


int base :: get_number()
{
        return number;
}

class derived : public base
{

};


void main()
{
        derived d;

        // First checks class derived , then class base
        cout << d.get_number();

        // Goes directly to class base
        cout<< d.base ::get_number();
}
```

Output:

```
        0
        0.
```

Using derived instances to access function members without the base class name and scope resolution operator, the compiler resolves the address by checking :

- Does the member function have scope in the derived class ? if so, use it ; else
- Does the member have scope in the parent class ? if so, use it ; else
- Does the member have scope higher up in the hierarchy ? if so use it; else
- Does the member have accessible cope higher up in the hierarchy? If so use it; else
- Does the member exist in the file scope ? if so use it ; else generate a compilation error.

Inheritance is the most powerful use of class and is also, probably, the most important concept of OOP. It is the mechanism using which reusability of code is achieved. The facility to use the existing tested code reduces the time and cost of writing it all over again. This session introduced the concept of inheritance. The next session covers more suitable aspects of inheritance

Hiding overloaded functions

We cannot overload a base class function by redefining it in a derived class with a different argument list. Consider examples to see if same function name exists in base as well as derived classes.

e.g.

```
	class base
	{
		public :
			void f(int n )
			{
				cout << " n = " << n;
			}
	};

	class derived : public base
	{
		public :a
			void f(float n )
			{
				cout << " n = " << n;
			}
	};

	void main()
	{
		derived d;
		d.f(1);
	}
```

Output :

```
	n = 1.000000
```

Even though we have passed an integer, the compiler uses the function of the derived class because it is the one, compiler knows about. If there is no convincible match within the scope of the derived class, the inherited members from the class will not be examined. The following example shows this.

```
	class base
	{
		public :
			void f(char *ptr)
			{
				cout << " ptr = " << ptr;
			}
	};

	class derived : public base
	{
		public :
			void f(float n )
			{
```

```
                        cout << " n = " << n;
                }
 };

void main()
{
        derived d;
        d.f("A");
}
```

Output:

```
        'Can not convert 'char*' to 'double'
```

However, we can still access the hidden members from the base class using the scope resolution operator. So in the main if we call

```
        d.base::f("A");
```

The output will be :

```
        ptr = A
```

Constructor and Destructor function with derived classes

If there are constructors involved in the base class and the derived class, the compiler automatically calls both of them. This happens because as soon as the derived class constructor gets control and establishes formal arguments, the base class constructor will be called immediately, i.e., before the derived class initialization list is honored.

Syntax in the derived class base/members initialization list :

- The base class name (name of base constructor)
- A list of arguments between parentheses, as in any normal function call. To explicitly invoke the base class default constructor , leave the argument list empty.

Remember the base class constructor always is called first.

```
class base
{
        public :
                base(int n=0);
                ~base();
                int get_base_number();

        protected:
                int base_number;
};


base::base(int n=0)
{
        base_number = n;
        cout << "base constructor ";
}

base::~base()
{
        cout << "base destructor ";
}

int base::get_base_number()
{
        return base_number;
}

class derived : public base
{

  public :
        derived(int b=0, int d =0): base(b),derived_number(d);
        ~ derived();
        int get_derived_number();

   private :
        int derived_number;
};

derived::derived(int b=0, int d =0): base(b),derived_number(d)
{
        cout << " derived constructor ";
}

derived::~ derived()
{
        cout << "derived destructor ";
}


int derived::get_derived_number()
{
        return derived_number;
}


void main()
{
        derived d(1,2);
        cout < " d= " << d.get_base_number()<<",";
        cout << d.get_derived_number();
}
```

Output:

```
        Base constructor
        Derived constructor
        D = 1,2
        Derived  destructor
        Base destructor
```

As can be seen from the above example, the constructor for the base class is invoked before invoking the constructor for the derived class . This is like building the first storey of the building before proceeding for the second. With destructors, it is the other way round. Destructors for the derived class are invoked first, so that they can clean up the mess done by the constructor of the derived class . Then the destructor for the base class is called. It is like demolishing the top floor of a building before going for the lower one. A destructor for the derived class is to be defined only if its constructor allocates memory. Otherwise it can be just an empty function .

Object Initialization

An object of a derived class can be initialized to an object of a base class . If both the classes have same data members , then no specific constructor needs to be defined in the derived class . It uses the constructor of the base class . An object of a base class can be assigned to the object of the derived class , if the derived class doesn't contain any additional data members . However, if it does , then the assignment operator will have to be overloaded for the same.

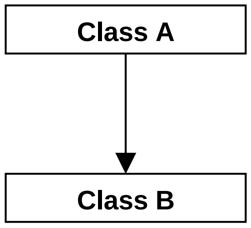
Different Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Single Inheritance : In "single inheritance," a common form of inheritance, classes have only one base class.

```
class  Aclass
{
        :
        :
};

class  Bclass : public Aclass
{
        :
        :
};
```

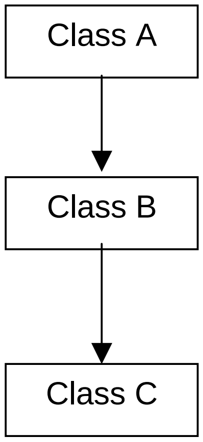


Multilevel Inheritance : In multilevel inheritance there is a parent class , from whom we derive another class . now from this derived class we can derive another class and so on.

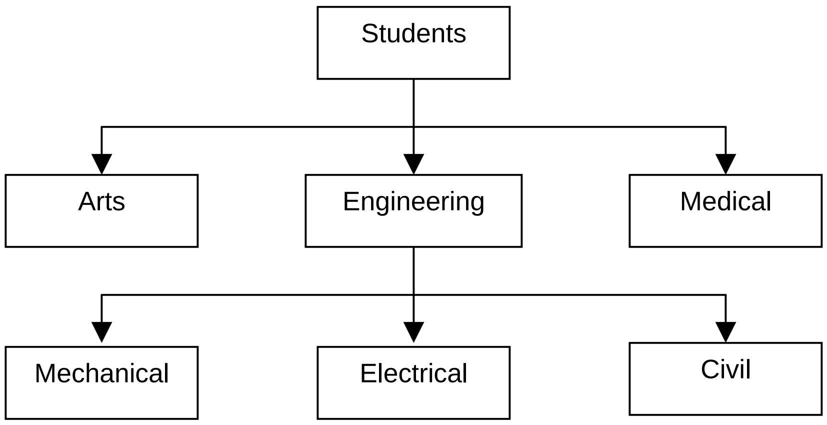
```
class  Aclass
{
        :
        :
}

class  Bclass : public Aclass
{
        :
        :
}

class  Cclass : public Bclass
{
        :
        :
}
```

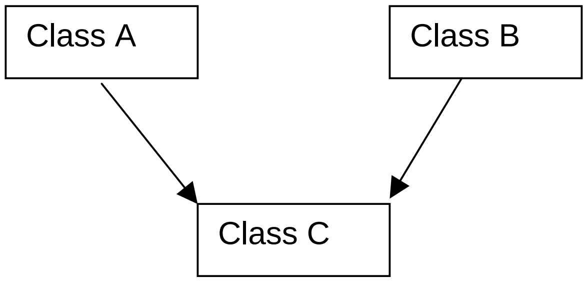


Hierarchical Inheritance : One important application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level. As an example, figure given below shows a hierarchical classification of students in a university.



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.

Multiple Inheritance : Multiple inheritance , as the name suggests , is deriving a class from more than one class . The derived class inherits all the properties of all its base classes. Consider the following example :



```
class Aclass
{
        :
        :
};
class  Bclass
{
        :
        :
};

class Cclass : public Aclass , public Bclass
{

        private :
                :
                :
        public  :

                Cclass(...) : Aclass (...), Bclass(...)
                {
                };
};
```

The class Cclass in the above example is derived from two classes – Aclass and Bclass – therefore the first line of its class definition contains the name of two classes, both publicly inherited. Like with normal inheritance , constructors have to be defined for initializing the data members of all classes. The constructor in Cclass calls constructors for base classes. The constructor calls are separated by commas.

Problems With Multiple Inheritance

The following example presents a problem with multiple inheritance.

```
        class Aclass
        {
                public :
                        void put()
                        {
                                    :
                        }
        };


        class  Bclass
        {
        public :
                    void put()
                    {
                                :
                    }
        };

        class Cclass : public Aclass , public Bclass
        {
                    public  :
                                :
                                :

        };

        void main()
        {
                    A objA;
                    B objB;
                    C objC;
                    objA.put();     // From Class A
                    objB.put();         // From class B
                    objC.put();         // AMBIGUOUS –RESULTS IN ERROR
        }
```

The above example has a class C derived from two classes A and B. Both these classes have a function with the same name – put(), which assume, the derived class does not have. The class C inherits the put() function from both the classes. When a call to this function is made using the object of the derived class , the compiler does not know which class it is referring to. In this case, the scope resolution operator has to be used to specify the correct object . Its usage is very simple. The statement giving an error from the above example has to be replaced with the following :

```
        objC.A::put();     // for class A
        objC.B::put();     // for class B
```

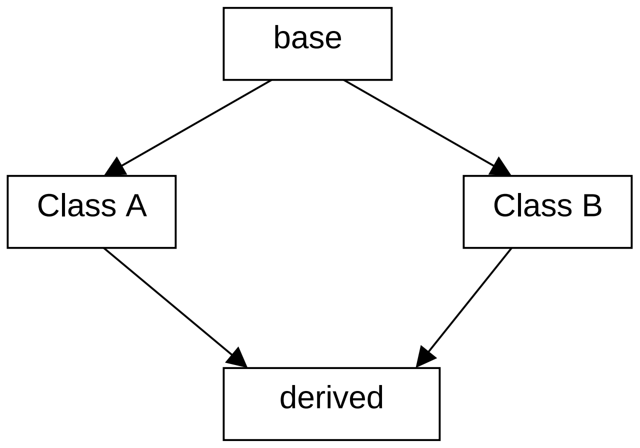
Hybrid Inheritance : Inheritance is an important and powerful feature of OOP. Only the imagination of the person concerned is the limit. There are many combinations in which inheritance can be put to use. For instance, inheriting a class from two different classes, which in turn have been derived from the same base class .

```
        class base
        {
                    :
                    :
        };

        class  Aclass : public base
        {
                    :
                    :
        };
```

```
class  Bclass : public base
{
        :
        :
};

class  derived : public Aclass, public Bclass
{
        :
        :
};
```



Aclass and Bclass are two classes derived from the same base class . The class derived has a common ancestor – class base. This is multiple inheritance with a common base class . However, this subtlety of class inheritance is not all that simple. One potential problem here is that both, Aclass and Bclass, are derived from base and therefore both of them, contains a copy of the data members base class. The class derived is derived from these two classes. That means it contains two copies of base class members – one from Aclass and the other from Bclass. This gives rise to ambiguity between the base data members. Another problem is that declaring an object of class derived will invoke the base class constructor twice. The solution to this problem is provided by virtual base classes.

Virtual Base Classes

This ambiguity can be resolved if the class derived contains only one copy of the class base. This can be done by making the base class a virtual class. This keyword makes the two classes share a single copy of their base class . It can be done as follows :

```
class base
{
        :
        :
};


class  Aclass : virtual public base
{
        :
        :
};

class  Bclass : virtual public base
{
        :
        :
};

class  derived : public Aclass, public Bclass
{
        :
        :
};
```

This will resolve the ambiguity involved.

Abstract Classes

Abstract classes are the classes, which are written just to act as base classes. Consider the following classes.

```
class base
{
        :
        :
};

class  Aclass : public base
{
        :
        :
};

class  Bclass : public base
{
        :
        :
};

class  Cclass : public base
{
        :
        :
};


void main()
{
        Aclass objA;
        Bclass objB;
        Cclass objC;
        :
        :
}
```

There are three classes – Aclass, Bclass, Cclass – each of which is derived from the class base. The main () function declares three objects of each of these three classes. However, it does not declare any object of the class base. This class is a general class whose sole purpose is to serve as a base class for the other three. Classes used only for the purpose of deriving other classes from them are called as abstract classes. They simply serve as base class , and no objects for such classes are created.

Overriding

A derived class can use the methods of its base class(es), or it can override them. The method in the derived class must have the same signature and return type as the base class method to override. The signature is number and type of arguments and the constantness (const, non- const) of the method. When an object of the base class is used, the base class method is called. When an object of the subclass is used, its version of the method is used if the method is overridden. Note that overriding is different from overloading. With overloading, many methods of the same name with different signatures (different number and/or types of arguments) are created. With overriding, the method in the subclass has the identical signature to the method in the base class. With overriding, a subclass implements its own version of a base class method. The subclass can selectively use some base class methods as they are, and override others. In the following example, the speak method of the Pet class will be overridden in each subclass.

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

class Pet
{
public:
        void speak();
};

void Pet::speak()
{
   cout << "Growl";
}

class Rat: public Pet
{
public:
        void speak();
};

void Rat::speak()
{
   cout << "Rat noise";
}

class Cat: public Pet
{
public:
        void speak();
};

void Cat::speak()
{
   cout << "Meow";
}

int main()
{
        Pet peter;
        Rat ralph;
        Cat chris;

        peter.speak();
        ralph.speak();
        chris.speak();

        return 0;
}
```

Notice that each subclass implements and used its own speak method. The base class speak method is overridden. Also, remember that the return type and signature of the subclass method must match the base class method exactly to override. Another important point is that if the base class had overloaded a particular method, overriding a single one of the overloads will hide the rest. For instance, suppose the Pet class had defined several speak methods.

```
void speak();
void speak(string s);
void speak(string s, int loudness);
```

If the subclass, Cat, defined only

```
void speak();
```

Then speak() would be overridden. speak(string s) and speak(string s, int loudness) would be hidden. This means that if we had a cat object, fluffy, we could call:

```
    fluffy.speak();
```

But the following would cause compilation errors.

```
    fluffy.speak("Hello");
    fluffy.speak("Hello", 10);
```

Generally, if you override an overloaded base class method you should either override every one of the overloads, or carefully consider why you are not.

Pointers to Objects

Pointers can be made to point to objects as well. The usage of pointer with classes is exactly like that of its usage with structures. As with structures the structure pointer (->) can be used to access members of the class . The following example illustrates its usage.

```
class  Person
{
        public :
                void getname()
                {
                        cout << “enter name “<< endl;
                        cin >> name;
                }

                void putname()
                {
                        cout << “ name = “ << name ;
                }
        private :
                char name[50];
};

void main()
{
        Person Bill;
        Bill.getname();
        Bill.putname();

        Person *Bush;
        Bush = new Person;
        Bush->getname();
        Bush->putname();
        Delete Bush;
 }
```

Pointers to objects are generally used to form complex data structures like linked lists and trees. However, with inheritance, pointers can sometimes become problematic. The following example illustrates yet another problem with class inheritance .

```
class Shape
{
        public :
                void print()
                {
                        cout << “ I am a Shape “ << endl;
                }
};
```

```
class Triangle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a Triangle " << endl;
                }
};

class  Circle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a  Circle " << endl;
                }
 };


void main()
{
        Shape S;
        Triangle T;
        Circle C;

        S.print();
        T.print();
        C.print();

        Shape *ptr;
        ptr = &S;
        ptr -> print();

        ptr = &T;
        ptr -> print();

        ptr = &C;
        ptr -> print();

}
```

The example contains a base class called Shape. Two other classes – Triangle and Circle – are derived from the base class . The program declares three objects of the three classes. The first part of the program calls print() function related to each of these classes. In the second part the program declares a pointer to the base class Shape and assigns address of each of the object declared earlier to this pointer.

What is the objective of assigning addresses of the derived classes to a pointer of the base class ? Further, how can the compiler allow such a thing ? Well, the compiler does not give an error on assigning address of a derived class to a pointer of the base class, because objects of the derived classes are type-compatible with pointers of base classes. This answers the second question. As far as the first question is concerned, the answer is polymorphism. That is, if the pointer points to the base class Shape, then the print() function from base class is invoked. Whereas, if the pointer is pointing to any of the derived class Triangle or Circle then the print()function from these respective classes is invoked. That is what polymorphism is all about – giving different meanings to the same thing.

The output of the program is given below:

```
I am a Shape
I am a Triangle
```

```
I am a Circle

I am a Shape
I am a Shape
I am a Shape
```

As can be seen, the first part works fine. But , for the second part, for each of the calls with different addresses, the function from the base class Shape is executed. What happens, here , is that the compiler ignores the contents of the pointer. Instead, to determine which function should be invoked, the compiler chooses the function from the same class as that of the pointer. But , this is not polymorphism. The keyword virtual again comes into rescue, but in a different context.

Virtual Functions

The keyword virtual was earlier used to resolve ambiguity for a class derived from two classes, both having a common ancestor. These classes are called virtual base classes. This time it helps in implementing the idea of polymorphism with class inheritance . The function of the base class can be declared with the keyword virtual. The program with this change and its output is given below.

```
class Shape
{
        public :
                virtual void print()
                {
                        cout << " I am a Shape " << endl;
                }
};

class Triangle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a Triangle " << endl;
                }
};

class  Circle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a  Circle " << endl;
                }
};

void main()
{
        Shape S;
        Triangle T;
        Circle C;
        S.print();
        T.print();
        C.print();

        Shape *ptr;
        ptr = &S;
        ptr -> print();
        ptr = &T;
        ptr -> print();

        ptr = &C;
        ptr -> print();
}
```

The output of the program is given below:

```
I am a Shape
I am a Triangle
I am a Circle

I am a Shape
I am a Triangle
I am a Circle
```

Now, the output of the derived classes are invoked correctly. When declared with the keyword virtual , the compiler selects the function to be invoked, based upon the contents of the pointer and not the type of the pointer. This facility can be very effectively used when many such classes are derived from one base class . Member functions of each of these can be ,then, invoked using a pointer to the base class .

Pure Virtual Functions

As discussed earlier, an abstract class is one, which is used just for deriving some other classes. No object of this class is declared and used in the program. Similarly, there are pure virtual functions which themselves won't be used. Consider the above example with some changes.

```
class Shape
{
        public :
                virtual void print() = 0; // Pure virtual function
};

class Triangle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a Triangle " << endl;
                }
};

class  Circle : public Shape
{
        public :
                void print()
                {
                        cout << " I am a  Circle " << endl;
                }
 };



void main()
{
        Shape S;
        Triangle T;
        Circle C;
```

```
            Shape *ptr;

            ptr = &T;
            ptr -> print();

            ptr = &C;
            ptr -> print();
    }
```

The output of the program is given below:

```
    I am a Triangle
    I am a Circle
```

It can be seen from the above example that , the print() function from the base class is not invoked at all . even though the function is not necessary, it cannot be avoided, because , the pointer of the class Shape must point to its members.

Object oriented programming has altered the program design process. Exciting OOP concepts like polymorphism have given a big boost to all this. Inheritance has further enhanced the language. This session has covered some of the finer aspects of inheritance. The next session will resolve some finer aspects of the language.

Virtual Destructors

Ask any programmer, he'll immediately reply saying "A destructor is a member function of a class, which gets called when the object goes out of scope". This means all clean ups and final steps of class destruction are to be done in destructor. A virtual function is something which helps a derived class in overriding the implementation of a functionality of a base class.

The order of execution of destructor in an inherited class during a clean up is like this.

1. Derived class destructor
2. Base class destructor

A difference between a destructor (of course also the constructor) and other member functions is that, if a regular member function has a body at the derived class, only the version at Derived class gets executed. Whereas in case of destructors, both derived as well as base class versions get executed.

Now turning our attention to why a destructor has to be virtual, the reason is that we, programmers are very smart. We'll do days and nights of work to inherit and extend the functionality of an existing class which is being used, and say that we don't want to change the implementation/interface just for the sake of a new entrant. Let me explain this with an example.

```
#include <iostream.h>
class Base
{
public:
                  Base(){ cout<<"Constructor: Base"<<endl;}
                 ~Base(){ cout<<"Destructor : Base"<<endl;}
};
class Derived: public Base
{
            //Doing a lot of jobs by extending the functionality
public:
             Derived(){ cout<<"Constructor: Derived"<<endl;}
            ~Derived(){ cout<<"Destructor : Derived"<<endl;}
};
void main()
{
      Base *Var = new Derived();
      delete Var;
}
```

Try executing this code, you'll see the difference. To our observation, the constructors are getting called in the proper order. But to the dread of a programmer of a large project, the destructor of the derived class was not called at all.

This is where the virtual mechanism comes into our rescue. By making the Base class Destructor virtual, both the destructors will be called in order. The following is the corrected sample.

```
#include <iostream.h>
class Base
{
public:
        Base(){ cout<<"Constructor: Base"<<endl;}
        virtual ~Base(){ cout<<"Destructor : Base"<<endl;}
};

class Derived: public Base
{
//Doing a lot of jobs by extending the functionality
public:
        Derived(){ cout<<"Constructor: Derived"<<endl;}
        ~Derived(){ cout<<"Destructor : Derived"<<endl;}
};

void main()
{
        Base *Var = new Derived();
        delete Var;
}
```

Note:
There is one more point to be noted regarding virtual destructor. We can't declare pure virtual destructor. Even if a virtual destructor is declared as pure, it will have to implement an empty body (at least) for the destructor.

Polymorphism – Runtime & Compile time polymorphism

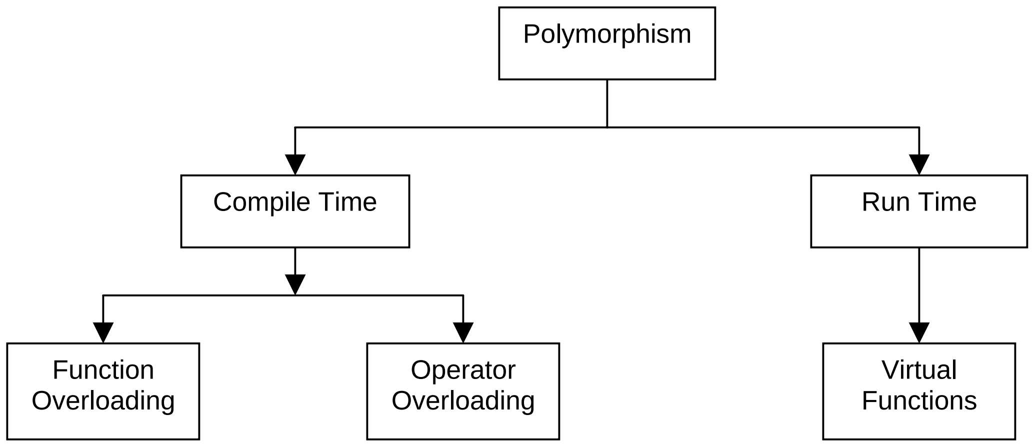
Polymorphism is one of the crucial features of OOP. It simply means ‘one name, multiple forms’. Function Overloading & Operator Overloading are examples of polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to 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 linking**. Also known as **compile time polymorphism**.

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();
}
```

```
class B : public A
{
        int y;
        public:
                void show();
}
```

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. To solve this problem, it would be nice if the appropriate member function could be selected while the program is running. This is known as **runtime polymorphism**. C++ supports a mechanism known as virtual function to achieve runtime polymorphism. 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 runtime. Object pointers and virtual functions are used to implement dynamic binding.



Exercises

1. WAP to add 2 complex number using OOT(Operator Overloading techniques).

2. WAP to add 2 times using OOT and display the resultant time in watch format.

3. WAP to add, subtract and multiply 2 matrices using OOT. Sort an array of objects. Each object has a string as a member variable. Overload >= or <= operators to compare the two strings. { make use of constructors and destructors whenever possible }

4. WAP to create a class called DATE . Accept 2 valid dates in the form of dd/mm/yyyy. Implement the following by overloading the operators **–** and **+** . Display the result after every operation.

   a) no_of_dasy = d1 – d2, where d1 and d2 are DATE objects;

      d1 > = d2 ; and no_of_days is an integer.

   b) d1 = d1 + no_of_days - where d1 is a DATE object and
      no_of_days is an integer.

5. Modify the matrix program (program 3) slightly. Overload == operator to compare 2 matrices to be added or subtracted. i.e., whether the column of first and the row of second matrix are same or not.

```
if(m1==m2)
 {
    m3=m1+m2;
    m4=m1-m2;
 }
```

```
else
        display error;
```

6. WAP to concatenate 2 strings by using a copy constructor.

7. Create a class DateClass with day, month and year as its members. Write an overloaded function for the minus sign. The first function returns the difference between two dates in terms of days.
   e.g.

```
DateClass date1, date2;
int diffdays;

diffdays = date1 – date2;
```

   The second overloaded function is written which returns a date occurring a given number of days ago.
   e.g.

```
DateClass newdate, somedate;
int days;

newdate = somedate – days;
```

8. WAP to create a class called as COMPLEX and implement the following by overloading the function ADD which returns a complex number.
   a) ADD(s1, s2) – where s1 is an integer (real) and s2 is a COMPLEX number.
   b) ADD(s1, s2)- where s1 and s2 are COMPLEX numbers.

9. Create a class drugs containing encapsulated data for medicine name, whether solid or liquid, price and purpose of use. From this class derive two classes, Ayurvedic and Allopathic. The class Ayurvedic should additionally store data on the herbs used, association to be used (whether honey or water). The class Allopathic should additionally include data on the chemicals used and the weight in milligrams. The classes should contain constructors and destructors. They should contain functions to accept data and display the data. The main() should test the derived classes.