Friend Function

The private member cannot be accessed from the functions that are not member to the class. However, there could be a situation where two classes need to share a particular function.

In such situations C++ allows the common function to be made *friendly* with both the classes thereby allowing the function to have access to the private data of these classes.

Such a function need not be a member of any of these classes.

To make an outside function "*friend*" to a class it will be declared as in the following example:

Example 1:

```
class ABC
{    public:
          friend void xyz ( ) ;  //declaration or prototype
 }
```

The function declaration should be preceded by the keyword *friend*.

The function definition does not use either the keyword friend or the scope operator (::).

A friend function has certain special characteristics:

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

Example 2:

```
class sample
{  private:  int x , y  ;
   public:
        void set ( int a , int b) {  x = a  ;  y  =  b  ;  }
        friend float mean (sample) ;
} ;
float  mean ( sample s )      // note that there is no (::) operator
{  return ( s.x  +  s.y  )  /  2.0  ;  }    // it uses s.x  instead of x

 main ( )
{ sample  a   ;
   a . set (20 , 45) ;
  cout << "mean value = " << mean (a) << endl  ;
 }
```

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

Example 3:

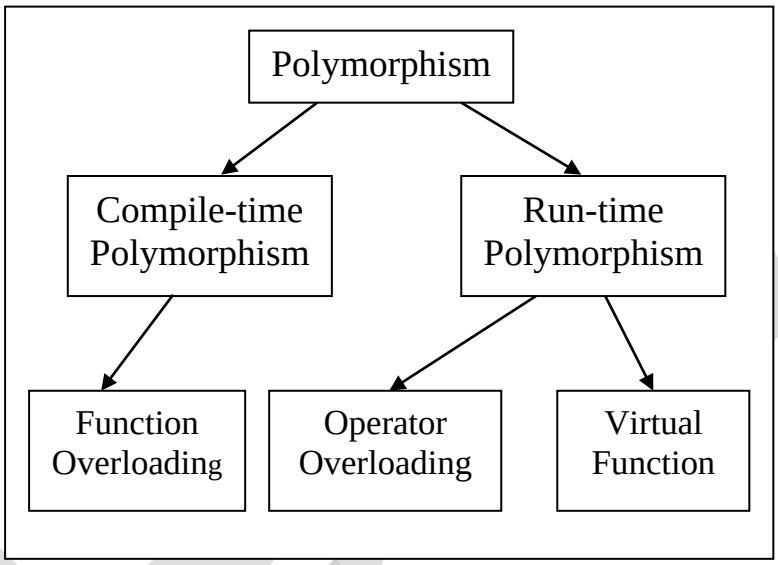
```
 class X
{   ….
    …..
    int  fun1 ( )  ;   //member function of X
 } ;

  class Y
 {  …….
     ……
    friend  int  X :: fun1( )  ; //The function fun1( ) is a member of class X and a friend of Y
     ……
  }  ;
```

We can also declare all the member function of one class as the friend functions of another class. In such case, the class is called a *friend class*.

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means "one name, multiple forms".



Function Overloading

Several different functions can be given the same name. Function overloading is one of the most powerful features of C++ programming language. It forms the basis of polymorphism (compile-time 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 binding*** or ***static linking***. Also known as ***compile time polymorphism***, early binding simply means that an object is bound to its function call at compile time.

Function Overloading: is the process of using two or more functions with the same name but differing in the signature (the number or type of arguments or both).

To avoid ambiguity, each definition of an overloaded function must have a unique signature.

All of you know that we cannot have two variables of the same name, but can we have two Functions having the same name. Such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

In overloaded functions, the function call determines which function definition will be executed.

The Advantages of Overloading are:

- It helps us to perform same operations on different data-types without having the need to use separate names for each version.

- The use of overloading may not have reduced the code complexity /size but has definitely made it easier to understand and avoided the necessity of remembering different names for each version function which perform identically the same task.

Example1: We have two functions with the same name: *calc*
We have different signatures: *(int) , (int , int)*
Overloading Functions differ in terms of *number of parameters*

```
#include <iostream.h>
class arith
{  public:
     void calc ( int num1)
     { cout << " Square of a given number: " << num1 * num1 << endl  ; }

    void calc ( int num1, int num2 )
     { cout << " Product of two numbers: " << num1 * num2  << endl ;  }
} ;
main ( )
{ arith a;
  a . calc ( 5 ) ;       // Based on the arguments we use when we call the calc function, the
  a . calc ( 6 , 7 ) ;  // compiler decides witch function to use at the moment we call the function
}
```

Example 2: Overloading Functions differ in terms of *type of parameters*

```
# include <iostream.h>
 int  func ( int  i ) ;
 double  func ( double  i ) ;

 main ( )
 {    cout <<  func ( 10 ) ;   //  func(int i)  is called
      cout <<  func (10.24 ) ;  //  func (double i) is called
 }
 int  func ( int  i )
 {       return  i  + 5 ;      }

 double  func ( double  i )
  {    return  i  * 8  ;      }
```

Example 3: Is the program below, valid ?

```
#include<iostream.h>
 int  func ( int  i )
 {      return i   ;       }
 double  func ( int  i )
 {
  return i;
 }

 void main ( )
 {      cout << func(10) ;

        cout << func(10.201) ;
 }
```

No, because you can't overload functions if they differ **only** in terms of the data type they return.

H.W. Write a program to use three functions (sum) to find the summation of two numbers.

Operator Overloading

C++ tries to make user-defined data types behave in much the same way as the built-in types. For instance, C++ permits to **add** two variables of user defined types with

the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with special meaning for a data type. The mechanism of giving such special meaning to an operator is known as ***"operator overloading".***

All C++ operators can be overloaded (given additional meaning) except the following:
- class member access operator (**.**)
- scope resolsution operator (::).
- size operator (sizeof)
- conditional operator (? :)

When an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Although the semantic of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associatively. For example: the multiplication operator is higher precedence than the addition operator.

Defining Operator Overloading:

The general form of an operator function is

```
Return type classname :: operator op ( arguments-list )
{
      Function body

}
```

Operator is reserved word and (op) is the operator being overloaded, ***operator*** op is the function name.

Note that operator functions can be either member function or friend function.

A basic difference between them is that a friend function take two argument for binary operators and one for unary operator while the member function take one argument for binary operator and no argument for unary operator.

This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with the friend function.

The process of overloading involves the following steps:

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

Overloading Unary Operators

Example: A minus operator, when used as a unary, takes just one operand

```
void  space :: operator - ( )
{  x  =  - x  ;
   y =  - y  ;
   z =  - z  ;
}
void  space :: getdata ( int a  , int b  ,  int c )
{ x =  a  ;
  y =  b  ;
  z  =  c  ;      }
void  space :: display ( )
{
 cout << x <<  " " << y << " " << z <<  "\n"  ;    }
main ( )
{ space  s  ;
  s . getdata ( 10 , -20 , 30 )  ;
  cout  <<  " s : "  ;
  s . display ( )  ;
  s . operator - (  )  ;    //   or   -s ;
  cout  <<  " s :  "   ;
  s . display ( ) ;
}
```

Note that statement like: s2 = - s1 ;
Will not work because the function operator – ( ) does not return any value. It can work if the function is modified to return object.

Overloading Unary Operators using Friends

To overload a unary minus operator using a friend function as follow:

```
void operator  - ( space & ss )
{ ss . x  = -  ss . x  ;
  ss . y  = -  ss . y  ;
  ss . z  = -  ss  . z  ;
}
```

Overloading Binary Operators:

The following example explain how to perform overloading for + operator

```
class complex
{  float  x  ;
   float   y  ;
  public:
        complex ( )
          {   }

        complex ( float real  ,  float image )
         {  x  =  real  ;
            y =  image  ;    }

        complex  operator + ( complex C ) ;

       void display( )
         { cout << x << " , " << y << "\n" ;  }
 } ;
complex  complex :: operator + ( complex C )
{  complex temp;
   temp . x = x  +  C . x  ;
   temp . y = y  +  C . y  ;
   return  temp ;
}
main ( )
{ complex  c1 ,  c2  ,  c3  ;
  c1 = complex ( 2.5 ,  3.5  )  ;
  c2 = complex ( 1.6 ,  2.7 )  ;
  c3 = c1 + c2  ;    //  or c =c1.operator + ( c2 ) ;
  cout << "c1= "  ;  c1 . display ( )  ;
  cout << "c2 = "  ;  c2 . display ( )  ;
  cout << "c3 = "  ;  c3 . display ( )  ;
```

}

Overloading Binary Operators using Friends

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

1- Replace the member function declaration by the friend function declaration
friend complex operator + (complex , complex)
2- Redefine operator functions as follows:

```
complex operator + ( complex  a  , complex  b )
{ complex  temp  ;
  temp . x = a . x  + b . x  ;
  temp . y = b . y  +  b . y  ;
  return  temp  ;
}
```

In this case the statement

```
c3  =  c1  +  c2  ;
```

Is equivalent to

```
c3  = operator + ( c1 , c2 )  ;
```

Note that the friend and member functions give the same result. Why then these alternatives? Consider a situation where one operand of binary operator is object and the second is built in data type as shown below:

```
A  =  B + 2  ;   ( or A = B * 2  )
```

This will work in member function and A , B object of the same class.

But the statement:

```
A = 2 + B  ;   (or A = 2  * B )
```

Will not work, this is because the left hand operand which is responsible for invoking the member function should be object of the same class. *Friend* function solves this problem.

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects.

Sometimes, we don't know, at the time we write the program, how many objects we want to create. When this is the case we can use ***new*** to create objects while the program is running.

Object pointers are useful in creating objects at run time. We can also use the object pointer to access the public members of an object.

```
Item x;
Item* ptr;
Item *ptr=&x;
```

Ptr->show () ; it is equivalent to (*ptr)->show();

We also can create the objects using pointers and new operator as follows:

```
Item*ptr=new Item;
ptr->show ( ) ;
Item*ptr  = new Item[10]  ;
ptr [0] -> show ( ) ;
```

When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator.

The next program illustrates how to access an object given a pointer to it:

Example 1:

```
class distance
{   private: int  feet  ;
              float inches ;

    public:  void  getdist ( )
             { cin >>  feet  ;
               cin  >> inches  ;   }
             void  showdist ( )
             {  cout  << feet  <<  "\n"  <<  inches    ;     }
} ;
main ( )
{ distance  dist  ;
  distance * distptr  ;   // pointer to distance
  dist . getdist ( ) ;
  dist . showdist ( ) ;
```

```
  distptr  =  new  distance ;   // points to new distance object
  distptr  ->  getdist ( )  ;     // access object members with  -> operator
}
```

Note: that we can not refer to the member functions in the object pointed by distptr using the dot (.) membership-access operator, as in

distptr . getdist ( ) ; // will not work, *distptr* is not a variable

The dot operator requires the identifier on its left to be a variable. Since *distptr* is a pointer to a variable, we need another syntax.

One approach of dereference (get the contents of the variable pointed by) the pointer:

```
( *distptr ) . getdist ( )  ;   // ok but it is inelegant
```

The parentheses are necessary because the dot (.) has higher the indirection operator (*). An equivalent but more concise approach is by using the membership-access operator ->.

distptr -> getdist ( ) ; // better approach

Example 2:
An Array of pointer to object

```
# include  <iostream.h>
class item
{  int  code  ;      float   price  ;
   public:
         void  getdata ( int a  , float  b )
          { code  = a  ;    price = b  ;  }
          void  show ( )
         { cout <<  code  <<  endl  ;
           cout <<  price  <<  endl  ;   }
} ;

const  int  size  =  2  ;
main ( )
{ item *p  =  new  item [size]  ;
```

```
  item *d  = p ;  // another pointer to first location in item in order not to loss the address of location
   int  x  ,  i  ;
   float  y  ;

   for ( i = 0  ;  i <  size  ;  i ++  )
   { cout << "input code and price for item"  <<  i+1  ;
     cin >>  x  >>  y  ;
     p -> getdata ( x , y ) ;
     p ++  ;
   }

  for ( i = 0  ;  i < size  ;  i ++ )
   { cout  <<  "item: "  <<  i+1  <<  endl  ;
     d ->  show ( )  ;
     d ++  ;
    }
}
```

Pointers to Derived Class

It is means that a single pointer variable can be made to point to objects belonging to different classes, for example, if B is a base class to the derived class D then declaring a pointer pointing to the base class can also be used to point to the derived D.

```
B *ptr  ;
B b ;
D d ;
ptr = &b  ;
ptr = &d ; // This is perfectly valid because d is an object derived from the class B
```

Example 3:

```
# include <iostream.h>
class B
{  public :    void display ( )
                 { cout << " hello B "  <<  endl ;  }
} ;
class  D :  public  B
{ public  :
            void display()
```