Object-oriented Programming
Lab 04
Friend functions, Operator overloading
Version 1.0

Friend function

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends. Friends are functions or classes declared with the friend keyword.

A **friend function** is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may or may not be a member of another class.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

Let's see the example;

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area () {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

```
24
```

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of classCRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, **to conduct operations between two different classes**. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

Now consider there are two classes named ABC and XYZ. Each has one attribute and we want to find the maximum value of the attribute of the two objects of the classes. So let's see how friend function helps us to access two class objects in a single function.

```
// friend functions
#include <iostream>
using namespace std;

class ABC; //forward declaration
class XYZ{
    int x;
  public:
    XYZ(int a){
        x=a;
    }
    friend void max(XYZ ob1, ABC ob2);
};
class ABC{
    int x;
  public:
    ABC(int a){
        x=a;
    }
    friend void max(XYZ ob1, ABC ob2);
};
void max(XYZ ob1, ABC ob2){
    if(ob1.x>=ob2.x){
        cout<<ob1.x<<"is maximum";
    }
    else{
        cout<<ob2.x<<"is maximum";
    }
}
int main () {
  ABC ob1(10);
  XYZ ob2(20);
  max(ob2, ob1);
  return 0;
}
```

```
20 is maximum
```

Here function max() has argument from both the classes. It is only possible when we declare the function max() as friend of both the classes. Look, here max() is called from the main

function with without any interference of any object. That is, the function is not belongs to any class but friend of both the class and can access the private members of both the classes. As friend function does not invoke by any object, therefore we need to pass the objects that we want to process in the friend function as parameters.

A friend function has some special characteristics:

1. It is not in the scope of the class to which it is declared as friend.
2. And thus cannot be called using the object of the class.
3. Can be called like a normal function.
4. Cannot access member names directly as member function, need to have the object of that member names to access and thus pass through arguments.
5. Friend function can be declared either in the public or private part of a class without affecting its meaning.

Friend Class:

A **friend class** is a class that can access the private and protected members of a class in which it is declared as **friend**. This is needed when we want to allow a particular class to access the private and protected members of a class.

Function Class Example

```
#include <iostream>
using namespace std;
class XYZ {
private:
   int num = 11;
public:
   /* This statement would make class ABC
    * a friend class of XYZ, this means that
    * ABC can access the private and protected
    * members of XYZ class.
    */
   friend class ABC;
};

class ABC {
public:
   void disp(XYZ obj){
      cout<<obj.num<<endl;
   }
};
int main() {
   ABC obj;
   XYZ obj2;
   obj.disp(obj2);
   return 0;
}
```

```
11
```

Operator overloading

In the lesson on function overloading, you learned that you can create multiple functions of the same name that work differently depending on parameter type. Operator overloading allows the programmer to define how operators (such as +, -, ==, =, and !) should interact with various data types. Because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading.
Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list. The general form of an operator function is:

```
Return type classname :: operator op(arg_list){
      Function body
}
```

Remember, when an operator is overloaded its original meaning is not lost.
Almost any operator in C++ can be overloaded. The exceptions are: arithmetic if (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).
You can overload the + operator to concatenate your user-defined string class, or add two Fraction class objects together. You can overload the << operator to make it easy to print your class to the screen (or a file). You can overload the equality operator (==) to compare two objects. This makes operator overloading one of the most useful features in C++ simply because it allows you to work with your classes in a more intuitive way.
Before we go into more details, there are a few things to keep in mind going forward.

First, at least one of the operands in any overloaded operator must be a user-defined type. This means you cannot overload the plus operator to work with one integer and one double. However, you could overload the plus operator to work with an integer and a class type.

Second, you can only overload the operators that exist. **You cannot create new operators. For example, you could not create an operator ** to do exponents.**

unary operator overloading
The unary operators operate on a single operand and following are the examples of Uninary operators:
The increment (++) and decrement (--) operators.
The unary minus (-) operator.
The logical not (!) operator.
The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Now let’s see an example of unary operator overloading;

```
// unary operator overloading
#include <iostream>
using namespace std;

class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      void displayDistance(){
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      void operator- (){
         feet = -feet;
         inches = -inches;
      }
};
int main()
{
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // apply negation
   D1.displayDistance();    // display D1

   -D2;                     // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```

```
F: -11 I:-10
F: 5 I:-11
```

In the above program we create a operator function named operator –(), that actually inverts the private member for an object. When the statement –D1 and –D2 executed from the main function, then each time the operator function called and invert the member for each object.

Now let's see how the member operator function is called for –D1 and –D2 statements. Usually we know that a member function can be called only by the help of an object that is using . an object can call a member function like D2.displayDistance(); above. Then how –D1 and –D2 statements calling operator –() function? There is no explicit calling of operator –(). Well, in that case of operator function, when ever compiler find any operator function and find the same operator with any user defined type variable, then calls that operator function doing some conversion before calling that function.

In the above example, when compiler finds the statement –D1, then it converts the statement as **D1.operator-()** statement and then calls the operator-() for D1. Also the same case happens for –D2.

Now let's see how compiler converts different operator statements and calls the operator functions;

1. For unary operators, operator functions can be invoked as **obj.operator op()** for member function and **operator op(obj)** for friend function.
2. For binary operators, operator functions can be invoked as **obj1.operator op(obj2)** for member function and **operator op(obj1, obj2)** for friend function.

Therefore, for binary operator we must declare one argument in member operator op() function and two argument in friend operator op() function. And we only need to declare an argument for unary friend operator op() function.

The process of overloading involves the following steps:

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

Binary operator overloading

You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explain how addition (+) operator can be overloaded. Similar way you can overload subtraction (-) and division (/) operators.

```
// Binary operator overloading
#include <iostream>
using namespace std;

class Box
{
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
public:
   Box(double x, double y, double z){
    length=x; breadth=y; height=z;
   }
   Box(){length=0; breadth=0; height=0;}

   double getVolume(void)
   {
      return length * breadth * height;
   }
   // Overload + operator to add two Box objects.
   Box operator+(Box b)
   {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
// Main function for the program
int main( )
{
   Box Box1(6.0,7.0,5.0);    // Declare Box1 of
type Box
   Box Box2(12.0,13.0,10.0); // Declare Box2 of
type Box
   Box Box3;                 // Declare Box3 of
type Box
   Box3 = Box1 + Box2; // Add two objects
   cout << "Volume of Box3 : " << Box3.getVolume()
<<endl;
   return 0;
}
```

```
Volume of Box3 : 5400
```

In the above example, when compiler finds the statement Box3 = Box1 + Box2;, then it converts the statement as **Box1.operator +(Box2)** statement and then calls the operator+(Box2) for Box1. And finally the returned object is assigned to Box3. In the operator+() function, you can see this->length and etc. Here this is pointer to the calling object, that is it is pointer to the object Box1.

Now convert above member operator +() function as friend function and do the same binary addition.

References:

1. http://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)
2. http://www.cplusplus.com/doc/tutorial/classes/
3. http://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm
4. Object-oriented Programming with C++ by E Balagurusamy
5. http://www.learncpp.com