National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2021, Lab Manual - 09

Course Code: CL-217	Course : Object Oriented Programming Lab

Contents:

Introduction to Friend Function:

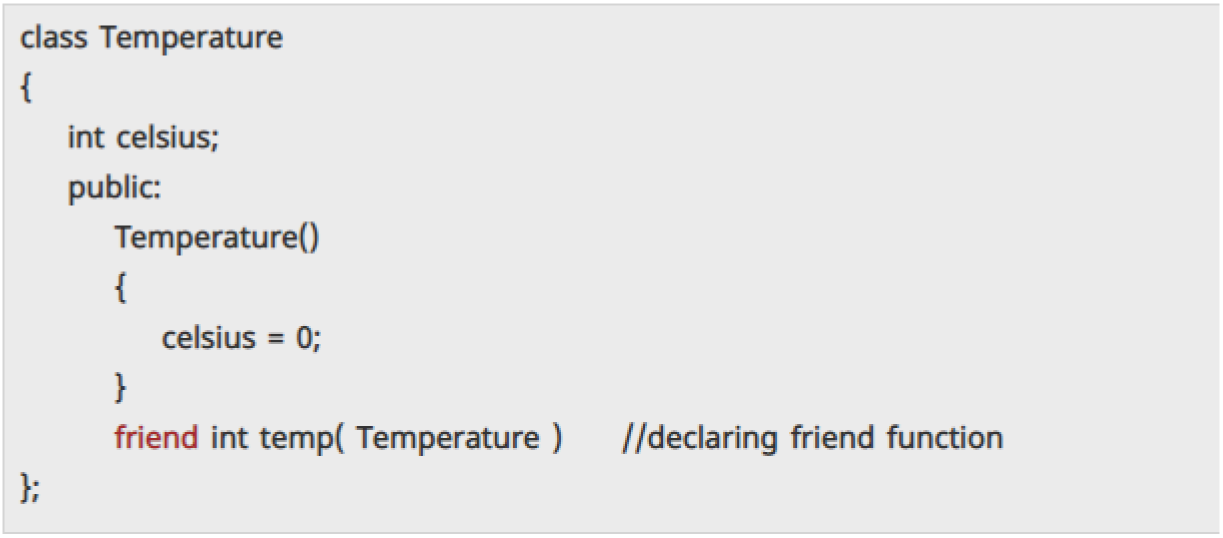
In object oriented programming, we have feature of hiding the data through access specifiers. Two of those access specifiers are private and protected. Private members cannot be accessed from outside of class while protected members can only be accessed through derived classes and cannot be accessed from outside of the class. Lets have an example :

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;

    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

In C++, **friend functions** are exception to this rule and these functions allow us to **access member functions from outside of the class** as well. The friend declaration can be placed anywhere in the class declaration



```
class Temperature
{
    int celsius;
    public:
        Temperature()
        {
            celsius = 0;
        }
        friend int temp( Temperature )      //declaring friend function
};
```

Friend Function in C++

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Example 1: Working of Friend Function

```
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
    private:
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance()
       {meter = 0;}

};
```

```
// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0; }
```

Output

```
  Distance: 5
```

Here, addFive() is a friend function that can access both **private** and **public** data members. Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use.
A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

Example 2: Add Members of Two Different Classes

```
// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

private:
        int numA;

public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}
```

```
         // friend function declaration
         friend int add(ClassA, ClassB);
};
class ClassB {
    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    private:
        int numB;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```

Output

```
Sum: 13
```

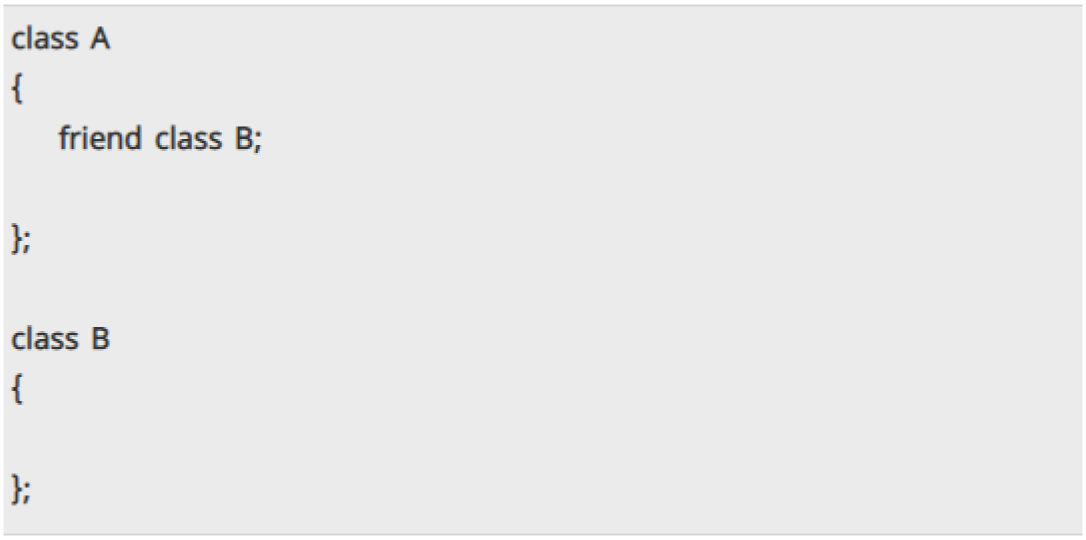
In this program, ClassA and ClassB have declared add() as a friend function. Thus, this function can access **private** data of both classes. One thing to notice here is the friend function inside ClassA is using the ClassB. However, we haven't defined ClassB at this point.

```
// inside classA
friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of ClassB in our program

```
// forward declaration
```

```
class ClassB;
```



```
class A
{
    friend class B;

};

class B
{

};
```

Friend Class in C++ :

We can also use a friend Class in C++ using the friend keyword. For example,

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA
   friend class ClassB;
   ... .. ...
}

class ClassB {
   ... .. ...
}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.

Since classB is a friend class, we can access all members of classA from inside classB.

However, we cannot access members of ClassB from inside classA.

<u>Example 3: Just like functions are made friends of classes, we can also make</u>

one class to be a friend of another class. Then, the friend class will have access to all the private members of the other class.

```
#include <iostream>
using namespace std;
class Square;
class Rectangle {
       int width, height;
public:
       Rectangle(int w, int h)
{      width=w; height=h; }

       void display() {
              cout << "Rectangle: " << width * height << endl;
       };
       void morph(Square &);
};

class Square {
       int side;
public:
       Square(int s)
       {side=s;}
       void display() {
              cout << "Square: " << side * side << endl;
       };
       friend class Rectangle;
};
void Rectangle::morph(Square &s) {
       width = s.side;
       height = s.side;}
int main () {
       Rectangle rec(5,10);
       Square sq(5);
```

```
        cout << "Before:" << endl;
        rec.display();
        sq.display();
        rec.morph(sq);
        cout << "\nAfter:" << endl;
        rec.display();
        sq.display();
        return 0; }
```

We declared **Rectangle** as a friend of **Square** so that **Rectangle** member functions could have access to the private member, **Square::side** In our example, **Rectangle** is considered as a friend class by **Square** but **Rectangle** does not consider **Square** to be a friend, so **Rectangle** can access the private members of **Square** but not the other way around.

Advantages/Applications of Friend Functions

- Windows will have many functions that should not be publicly accessible but that can be needed by a related class like WindowsManager so friend functions can be used.

- You want to operate on objects of two different classes then friend function can be useful as we observed in above examples otherwise with out friend function in this situation, your code will be long, complex and hard.

Disadvantages of Friend Functions

- One of the distinguishing features of C++ is encapsulation i.e. bundling of data and functions operating on that data together so that no outside function or class can access the data. But by allowing the friend functions or class to access the private members of another class, we actually compromise the encapsulation feature.

- In order to prevent this, we should be careful about using friend functions or class. We should ensure that we should not use too many friend functions and classes in our program which will totally compromise on the encapsulation.

Operator Overloading:

In C++, the purpose of **operator overloading** for an operator is to specify more than one meaning in one scope. It gives special meaning of an operator for a user-defined data type.

You can redefine most of the C++ operators with the help of operator overloading. To perform multiple operations using one operator, you can also use operator overloading.

Syntax of operator overloading:

To overload a C++ operator, you should define a special function inside the Class as follows:

```
class class_name
{
    ... .. ...
    public
       return_type operator symbol (argument(s))
       {
           ... .. ...
       }
    ... .. ...
};
```

Here is an explanation for the above syntax:

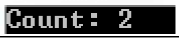
- The return_type is the return type for the function.
- Next, you mention the operator keyword.
- The symbol denotes the operator symbol to be overloaded. For example, +, -, <, ++.
- The argument(s) can be passed to the operator function in the same way as functions.

Example :

```
#include <iostream>
using namespace std;
class TestClass {
private:
      int count;
public:
      TestClass()
      {
      count=5;
```

```
        }
        void operator --() {
                count = count - 3;
        }
        void Display() {
                cout << "Count: " << count; }
};
int main() {
        TestClass tc;
        --tc;
        tc.Display();
        return 0;}
```

Output:



```
Count: 2
```

C++ Operators that cannot be Overloaded

There are four C++ operators that **can't be overloaded**.

They include:

- :: Scope resolution operator
- ?: ternary operator.
- . member selection operator
- .* member selection through a pointer to function operator.

C++ Operators that can be Overloaded

- Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||

+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Unary Operators Overloading:

The unary operators operate on a single operand and following are the examples of Unary 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--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, 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;
}
```

When the above code is compiled and executed, it produces the following result −

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

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

The increment (++) and decrement (--) operators are two important unary operators available in C++. Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

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

class Time {
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59

   public:
      // required constructors
      Time() {
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m) {
         hours = h;
         minutes = m;
      }
      // method to display time
      void displayTime() {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }

      // overloaded prefix ++ operator
      Time operator++ () {
         ++minutes;          // increment this object
         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }

      // overloaded postfix ++ operator
      Time operator++(int ) {

         // save the original value
         Time T(hours, minutes);

         // increment this object
         ++minutes;

         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }

         // return old original value
         return T;
      }
};

int main() {
   Time T1(11, 59), T2(10,40);

   ++T1;                    // increment T1
   T1.displayTime();        // display T1
   ++T1;                    // increment T1 again
   T1.displayTime();        // display T1

   T2++;                    // increment T2
   T2.displayTime();        // display T2
   T2++;                    // increment T2 again
   T2.displayTime();        // display T2
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

Binary Operators Overloading:

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

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

```
#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:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength(double len ) {
      length = len;
   }

   void setBreadth(double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const 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;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Relational Operators Overloading:

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

Live Demo

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }


      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
D2 is less than D1
```

Assignment Operator Overloading:

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

Function Call() Operator Overloading:

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Following example explains how a function call operator () can be overloaded.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // overload function call
      Distance operator()(int a, int b, int c) {
         Distance D;

         // just put random calculation
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches << endl;
      }
};
```

```
int main() {
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

Things to remember for Operator Overloading:

Here are rules for Operator Overloading:

- At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
- We can only overload the operators that exist and cannot create new operators or rename existing operators.
- You can make the operator overloading function a friend function if it needs to access the private and protected class members.
- Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using member function. Eg. Assignment overloading operator =, and function call() operator etc.

Lab Tasks:

Q#1

Write a program to overload decrement operator -- in such a way that when it is used as a prefix, it multiplies a number by 4 and when it is used as a postfix then it divides the number by 4.

Q#2

Write a program that will apply the concept of operator overloading on + operator to add the areas of shape1 and shape2. Name of class is “shape” while shape1 and shape2 are the objects of class shape. Use the same Area() function for both objects.

Q#3

Write the following program to show the working of friend class. Define two classes i-e “printClass” and “perimeter”. Perimeter class finds perimeter using length and breadth values where length and breadth values are private. Make “printClass” a friend of “perimeter” class. Once this is done, create an object in main class to calculate perimeter and pass this object to printClass to display perimeter.