**Academic Year: 2020**

**Semester: 2nd**

**Course Code: CS-241L**

**Course Title: Object Oriented Programming**

**CS-241L Object Oriented Programming Lab 08**

**Type of Lab: Open + Close Ended Weightage: 10%**

**CLO 2:** Apply composition, inheritance and polymorphism and language extension concepts to build classes.

|  |  |  |  |
| --- | --- | --- | --- |
| Student understand the concept of Friend function operator overloading | **Cognitive/Understanding** | CLO2 | Rubric A |

**Rubric A: Cognitive Domain**

**Evaluation Method: GA shall evaluate the students for Question according to following rubrics.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CLO** | **0** | **1** | **2** | **3** | **4** | **5** |
| CLO2 | Unable to understand and implement | Student understand Friend function concept | Student understand operator overloading concept | Implement friend function and operator overloading | Understand and implemented half problem sets | Understand and implemented complete problem sets |

# 

# **Lab 8**

**BS-Computer Science**

**Object Oriented Programming**

**Target:** Friend Function, this Pointer and Operator Overloading

## **Friend Function**

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

Friend function declaration can appear anywhere in the class. But a good practice would be where the class ends. An ordinary function that is not the member function of a class has no privilege to access the private data members, but the friend function does have the capability to access any private data members. The declaration of the friend function is very simple. The keyword friend in the class prototype inside the class definition precedes it. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

class className

{ ... .. ...

friend return\_type functionName(class name);

... .. ...}

return\_type functionName(argument/s)

{ ... .. ...

// Private and protected data of className can be accessed from this function because it is a friend function of className.

……………..}

The advantage of encapsulation and data hiding is that a non-member function of the class cannot access a member data of that class. Care must be taken using friend function because it breaks the natural encapsulation, which is one of the advantages of object-oriented programming. It is best used in the operator overloading.

**Addition of members of two different classes using friend Function**

#include <iostream>

using namespace std;

// forward declaration

class B;

class A {

private:

int numA;

public:

A(){

numA=10;}

// friend function declaration

friend int add(A, B);

};

class B {

private:

int numB;

public:

B() {

numb=20 }

// friend function declaration

friend int add(A , B); //it means that it is friend function of class A and B

};

// Function add() is the friend function of classes A and B

// that accesses the member variables numA and numB and objects are passing as arguments of friend function

int add(A objectA, B objectB)

{

int sum;

sum=objectA.numA+ objectB.numB

return sum;

}

int main()

{

A objectA;

B objectB;

cout<<"Sum: "<< add(objectA, objectB);

return 0;

}

## Class Problem Set:

1. **Write a C++ program to swap the values two integer members of different classes using friend function.**

**Pseudo code:**

|  |
| --- |
| **classA** |
| **Data Member:** value1 (integer) |
| **Methods:**  **setdata( ): (a type of void function used to take value of value1 (integer)**  **display()** : **(a type of void function used to display value of value1 before swaping.**  **exchange (class1 ,class2 ) :** Friend Function (swap and display values after swapping) |

|  |
| --- |
| **ClassB** |
| **Data Member:** value2 (integer) |
| **Methods:**  **setdata( ): (a type of void function used to take value of value2(integer)**  **display()** : **(a type of void function used to display value of value2 before swapping.**  **exchange (class1 ,class2) :**Friend Function (swap and display values after swapping) |

1. **Write a C++ program to swap the values two integer members of different classes using friend function (Use call by reference method).**
2. **Write a C++ program for addition of two complex numbers using friend function (use constructor function to initialize data members of complex class).**

|  |
| --- |
| **complex** |
| Int real  Int img |
| **Complex()**: (**default constructor with zero values of data members)**  **complex( real, img) : (Parametrized constructor with given inputs at the time of object creation)**  **complex sum( complex , complex)** :**(friend function )**  **show(complex ): display complex number of objects of class Complex.** |

Hint:The -> is called the **arrow operator**. It is formed by using the minus sign followed by a greater than sign. Simply saying: To access members of a structure, use the dot **operator**. To access members of a structure through a pointer, use the **arrow operator**.

# Static Members of a C++ Class

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.

The declaration of a static data member in the member list of a class is not a definition. You must define the static member outside of the class declaration, in namespace as by re-declaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

class X

{

public:

static int i;

};

int X::i = 0; // definition outside class declaration

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class X exist even though the static data member X::i has been defined.

**Problem Set:**

1. **Write a C++ that creates four objects with roll\_num, name and marks as attributes of class student. Each object of class must be assigned a unique roll\_number. (using roll\_number as static data member).**

|  |
| --- |
| **Student** |
| **Data Member:** name (character array), marks (float), roll\_num (int) |
| **Methods:**  **Student( ): (Constructor used to null and zero values of attributes).**  **Setdata() (a type of void function used to take value of attributes)**  **display()** : **(a type of void function used to display value of attributes )** |

## Static Function Members:

By declaring a function member as static, you make it independent of any object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

**‘this’ Pointer:** It is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

To understand ‘this’ pointer, it is important to know that how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.  
2. All access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share single copy of member functions.  
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?

Compiler supplies an implicit pointer along with the functions names as ‘this’. The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.

For a class X, the type of this pointer is ‘X\* const’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X \*const’

**Example:**

1) When local variable’s name is same as member’s name

|  |
| --- |
| #include<iostream.h>    /\* local variable is same as a member's name \*/  class Test  {  private:     int x;  public:     void setX (int x)     {         // The 'this' pointer is used to retrieve the object's x         // hidden by the local variable 'x'         this->x = x;     }     void print() { cout << "x = " << x << endl; }  };    int main()  {     Test obj;     int x = 20;     obj.setX(x);     obj.print();     return 0;  } |

Output:

x = 20

**2) To return reference to the calling object**

|  |
| --- |
| /\* Reference to the calling object can be returned \*/  Test& Test::func ()  {     // Some processing     return \*this;  } |

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

|  |
| --- |
| #include<iostream.h>    class Test  {  private:    int x;    int y;  public:    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }    Test &setX(int a) { x = a; return \*this; }    Test &setY(int b) { y = b; return \*this; }    void print() { cout << "x = " << x << " y = " << y << endl; }  };    int main()  {    Test obj1(5, 5);      // Chained function calls.  All calls modify the same object    // as the same object is returned by reference    obj1.setX(10).setY(20);      obj1.print();    return 0;  }  Output: x = 10 y = 20 |

**Problem Set**

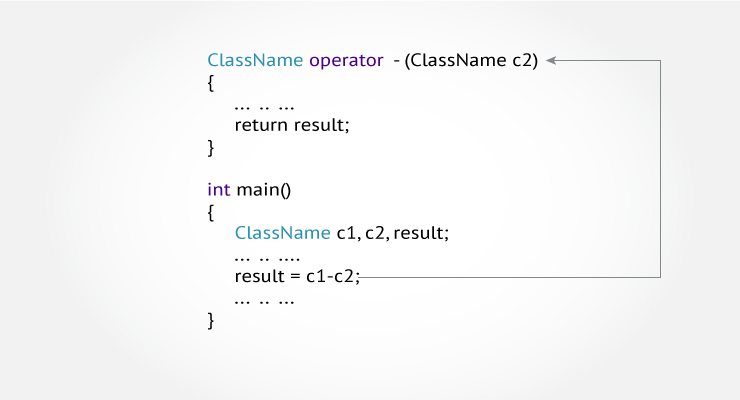
Predict the output of following programs. If there are compilation errors, then fix them.

|  |
| --- |
| #include<iostream.h>    class Test  {  private:    int x;  public:    Test(int x = 0) { this->x = x; }    void change(Test \*t) { this = t; }    void print() { cout << "x = " << x << endl; }  };    int main()  {    Test obj(5);    Test \*ptr = new Test (10);    obj.change(ptr);    obj.print();    return 0;  } |

**Operator overloading**

Operator overloading: is an important concept in C++. The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used. However, for user-defined types (like: objects), you can redefine the way operator works. For example: If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings. This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

To overload an operator, a special operator function is defined inside the class as:



* Here, returnType is the return type of the function.
* The returnType of the function is followed by operator keyword.
* Symbol is the operator symbol you want to overload. Like: +, <, -, ++
* You can pass arguments to the operator function in similar way as functions.

**Important points related to Operator Overloading:**

* Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
* Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
* Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
* There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .\* (member selection through pointer to function) and ?:(ternary operator).

**Example Overloading “<<” Operator**

#include< iostream.h>

#include< conio.h>

class time

{

int hr,min,sec;

public:

time()

{

hr=0, min=0; sec=0;

}

time(int h,int m, int s)

{

hr=h, min=m; sec=s;

}

**friend ostream& operator << (ostream &out, time &tm);** //overloading '<<' operator

};

ostream& operator<< (ostream &out, time &tm) //operator function

{

out << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";

return out;

}

void main()

{

time tm(3,15,45);

cout << tm;

}

**Example Overloading Relational Operators**

You can also overload Relational operator like == , != , >= , <= etc. to compare two user-defined object.

class time

{

int hr,min,sec;

public:

time()

{

hr=0, min=0; sec=0;

}

time(int h,int m, int s)

{

hr=h, min=m; sec=s;

}

friend bool operator==(time &t1, time &t2); //overloading '==' operator

};

bool operator== (time &t1, time &t2) //operator function

{

return ( t1.hr == t2.hr &&

t1.min == t2.min &&

t1.sec == t2.sec );

}

**Operator Overloading:** Feature in **C++** programming that allows programmer to redefine the meaning of an **operator** (when they operate on class objects) is known as **operator overloading**.

## Example Overloading ‘+’ Operator

#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;

}

**Class Problem Set**

1. Overload ‘-‘ and ‘\*’ operators to minus and multiply objects.
2. Overload = operator

**Home Tasks:**

1. Write a C++ program to perform matrix addition using operator overloading concept.
2. Write a C++ program that overloads arithmetic addition for concatenating two names of persons in person class and use comparator operator to come name of persons in person class.
3. Create a class called **Martix** that represents a 3x3 matrix. Create a constructor for initializing the matrix with 0 values. Create another overloaded constructor for initializing the matrix to the values sent from outside. Overload the + and += operators for addition of two matrices, == operator for checking the equality of two matrices, and \*operator for finding the product of the two matrices. Define all the member functions outside the class.
4. Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c) A member function **show** to display time in 11:59:59 format.

d) An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

e) Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.

f) Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

A **main()** programs should create two initialized **Time** objects and one that isn’t initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable. Check the functionalities of ++ and -- operators of this program for both pre- and post-incrementation.