|  |
| --- |
| **Programming Fundamentals** |
| **(CL214)** |
| **LABORATORY MANUAL** |
| **Spring 2021** |
| **C:\Users\Aamer\Desktop\nu-new.png**  **LAB 11** |
| ***Operator Overloading*** |
|  |

|  |  |  |
| --- | --- | --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | \_\_\_ |
| STUDENT NAME | ROLL NO | SEC |
|  | | |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
| LAB ENGINEER SIGNATURE & DATE | | |
| **MARKS AWARDED:** \_\_\_**/10** | | |
|  | | |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), ISLAMABAD** | | |
|  | | |

|  |  |
| --- | --- |
| **LAB 11** | **Operator Overloading** |

**Lab Objectives:**

1. To learn what is operator overloading and why is it required.
2. To learn how to overload unary, logical and binary operators.
3. To learn what are the restrictions on operator overloading.
4. To learn how to overload insertion and extraction operator.

**Software Required:**

Dev C++

**Introduction:**

# What is Operator Overloading and why is it Required?

Operator overloading is to allow the same operator to be bound to more than one implementation, depending on the types of the operands.

As you know that there are standard arithmetic operators in C/C++ for addition (+), subtraction (-), multiplication (\*) and division (/). We should only use these operators for their specific purposes. If we want to add two *int*, say *i* and *j*, the addition will take place in the following manner i.e. *i + j*. To add two double numbers, we use the same operator and write *d1 + d2*. We may add two floats with the help of the same operator as *f1 + f2*.

Similarly, other operations of -, \* and / on the primitive types (sometimes called as native or built-in types) can be employed. In other words, these operators are already overloaded for primitive types in C++. But these C++ operators cannot be used for classes and their objects. We have to write our own operator functions that can work with objects.

Let’s take an example of complex numbers. There are two parts of a complex number i.e. real and imaginary. As complex numbers are part of mathematical vocabulary, so the mathematical manipulations are done on them like addition, subtraction and multiplication. Suppose, we write our own class for complex numbers named Complex, but we can’t add two complex numbers *c1* and *c2* as *c1 + c2* because until now we don’t know how to write it. Although, we are able to write a function say *cadd*() to serve this purpose.

class Complex

{

float real ;

float imag ;

public:

Complex cadd(Complex c1, Complex c2 ) ;

}

It accepts two complex numbers as parameters and returns back the resultant complex number. But the usage of this function to add two complex numbers is generally clumsy. It gets more cumbersome and complex if we want to carry out cascading operations like *i + j + k*. It is better to use the standard operators of +, -, \* and / as they are more readable and elegant.

# Where is it Relevant to Apply?

Firstly, the operator overloading gets relevant whenever there is the application of the mathematical functions of addition, subtraction, multiplication and division. Complex number is one example of it.

Secondly, the operators are also used sometimes in case of non-mathematical manipulation. The example of String class to manipulate strings helps us understand it in a better way. The operator + can be used to concatenate two strings. Previously, we used *strcat()* function declared inside *string.h* header file to concatenate two strings. As compared to *strcat()*, the use of + to concatenate two strings is definitely easier and more readable. But there is a little bit cost associated with this process of operators overloading.

The cost is involved whenever we overload an operator. We have to write a function and make use of the operator semantics correctly while implementing the function. This means that the function written to overload + operator should do addition or concatenation of strings in case of String objects.

# Operators to Overload

There are two types of operators to overload:

* + Unary
  + Binary

Unary operators are the ones that require only one operator to work. Unary operators are applied to the left of the operand. For example, ^, & , ~ and ! .

Binary operators require two operands on both sides of the operator. +, -, \*, /, and > are examples of binary operators.

The complete list of C++ operators that can be overloaded is as follows:

+ - \* / % ^ &

| ~ ! = < > +=

-= \*= /= %= ^= &= |=

<< >> >>= <<= == != <=

>= && || ++ - - -> \* ,

[ ] ( ) new new[ ] delete delete[ ]

Let’s start with operator overloading mechanism. Consider an object date of the Date class. The data member day can be accessed as follows:

date.day = 2;

In this statement, the day data member of the date object is accessed and assigned value. This expression (*date.day*) is driven by the object name at left.

Similarly, while using operators, the statement like a + b is driven by the object at the left. In this case, + operator function for the object a will be called and b object is passed explicitly to the + operator function as an argument. The rules of function overloading are applied to the operator overloading. We cannot write two + operator functions with exactly identical parameters. Following the overloading rules, the two operator functions have to be different by the type or number of arguments.

The syntax of the prototype of the overloaded operator function is:

return-type operator-symbol (parameter-list);

operator is the keyword here. An example of this will be as follows:

Complex operator+ (Complex &);

We sometimes write only operator to refer to the operator function in our discussion.

To overload increment and decrement operator, use following operator definition:

|  |
| --- |
| operator++() ; //pre increment  operator++(int); //post increment |

# Restrictions on Operator Overloading

There are some restrictions on operator overloading:

* The following operators can’t be overloaded.

. : :: .\* ? sizeof

* The operator overloading functions for overloading (), [], -> or the assignment (=) operators must be declared as class members.
* The arity (number of operands) cannot be changed. If you are overloading an operator that requires two operands e.g. \*. It cannot be used as a unary operator that requires one operand.
* No new operators can be created. Like in Fortran language, we have \*\* as raise to the power (exponent) operator but this operator does not exist in C++. Therefore, it can’t be overloaded. Hence, only existing operators of C++ are used.
* Overloading can’t be performed for the built-in (sometimes called primitive or native) data types. For example, we cannot change how two *int*s are added. That means that operators are overloaded to use with defined data types like classes.
* Precedence of an operator cannot be changed. For example, the \* has higher precedence than +. This precedence cannot be changed.
* Associativity of an operator cannot be changed. If some operator is right associative, it cannot be changed to be left associative.

1. **Logical Operator:**

Logical operators are typically used with [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) (logical) values. When they are, they return a Boolean value. Most common operator are <, > and =, <=, >=, != and ==.

1. **Overloading the Stream Insertion (<<) and Extraction (>>) Operators**

The operator function that overloads the insertion operator, <<, or the extraction operator, >>, for a class must be a nonmember function of that class. The general syntax to overload the stream insertion operator, <<, for a class is:

|  |
| --- |
| friend ostream& operator<<(ostream&, const className&); |

Function definition is given by:

|  |
| --- |
| ostream& operator<<(ostream& osObject, const className& cObject)  {  //local declaration, if any  //Output the members of cObject.  //osObject << . . .  //Return the stream object.  return osObject;  } |

In this function definition:

* + Both parameters are reference parameters.
  + The first parameter—that is, osObject— is a reference to an ostream object.
  + The second parameter is usually a const reference to a particular class, because the most effective way to pass an object as a parameter to a class is by reference.

Similarly, the general syntax to overload the stream extraction operator, >> is:

|  |
| --- |
| friend istream& operator>>(istream&, className&); |

Function definition is given by:

|  |
| --- |
| istream& operator>>(istream& isObject, className& cObject)  {  //local declaration, if any  //Read the data into cObject.  //isObject >> . . .  //Return the stream object.  return isObject;  } |

In this function definition:

* + Both parameters are reference parameters.
  + The first parameter—that is, isObject—is a reference to an istream object.
  + The second parameter is usually a reference to a particular class. The data read will be stored in the object.
  + The function return type is a reference to an istream object.

1. **Examples of Operator Overloading**

Let’s take the complex numbers class Complex and define a + operator function.

We know that when we write the following line:

x = y + z;

y and z operands take part in the addition operation but there is no change in them due to this operation. This is the + operators functionality. The resultant is being assigned to the variable x. This is assignment operator’s functionality.

Now we will discuss a little bit about the assignment operator as well. Let’s say we write the following statement for two complex numbers c1 and c2.

c1 = c2 ;

Here c2 is being assigned to c1. Will this assignment work when we have not written any assignment operator function for complex number? Apparently, it looks that the statement will produce a compilation error (as there is assignment operator defined by us) but this is not true. Whenever, we write our own class and compile it, the compiler automatically generates a default assignment operator. The default assignment operator makes a member to member assignment. This works fine unless there is a pointer data member inside our class and that pointer is pointing to some data inside memory. For that case (when there is a pointer data member) we have to write our own assignment operator otherwise the default assignment operator works fine for us.

By definition of addition of complex numbers, we know that whenever two complex numbers are added, the real part of one number is added into the real part of other number. Similarly, the imaginary part of one number is added to the imaginary part of the other number. We also know that when a complex number is added to another complex number, the resultant is also a complex number consisting of real and imaginary parts.

This addition of real, imaginary parts and return of resultant complex number is the functionality of the + operator function we are going to write.

Another thing to decide for this + operator is whether this operator will be a member operator or a friend operator. Normally, operators are member operators but there are situations when they cannot be member operators. In case of member operator, following is the syntax of its prototype:

Complex operator + (parameter-list);

For member operator, the object on the left side of the + operator is driving this + operation. Therefore, the driving object on the left is available by this pointer to + operator function. But the object on the right is passed explicitly to the + operator as an argument.

We can define a member operator as under:

Complex Complex :: operator + ( Complex c)

{

Complex temp ;

temp . real = real + c. real ;

temp . imag = imag + c. imag ;

return temp ;

}

Let’s see this code line by line.

Line 1 indicates that the return type is Complex, it is an operator + function and it is accepting a *Complex* object by value as an argument. In line 3, a local *Complex* object is declared, called *temp*. In line 4, *real* part of the calling object (that is the one, driving) on the left of the + operator is being added to the *real* part of the object *c*, where *c* is passed as an argument. In line 5, *imag* part of the calling object (that is the one, driving) on the left of the + operator is being added to the *imag* part of the object *c*, where *c* is passed as an argument. In line 6, the *Complex* object *temp* containing the resultant of + operation is being returned by value. In our code, we can write something as:

Complex c1, c2, c3 ;

c3 = c1 + c2 ;

In the above statement ( *c3 = c1 + c2;* ), *c1* is the object that is calling or driving the

+ operator. *c2* object is being passed as an argument to the + operator. So *c1* and *c2* objects are added by the + operator and resultant *Complex* object containing the addition of these two numbers is returned back. That returned *Complex* object is assigned to the *c3* *Complex* object using the default assignment operator (that is created by the C++ compiler automatically).

What happens if we want to add a double number to a complex number? Like the following:

c3 = c1 + d ;

This + operation is driven by the *c1* object of *Complex* while double number *d* of type double is passed as argument. Therefore, our above written + operator is not useable for this operation of addition. We need to overload + operator for accepting a parameter of type double, i.e. we need to write another operator function. The definition of this newly overloaded + operator is:

Complex Complex :: operator + ( double d )

{ Complex temp ;

temp . real = real + d ;

temp . imag = imag ;

return temp ;

}

By now, you should have noticed that operator overloading and function overloading are quite similar.

When we write the following statement:

c3 = d + c1;

The operand on the left of + operator is a double number d. Therefore, this + operation should be driven by (called by) the double number. Until now, we have not written such an operator. Our previously written two + operators were driven by the Complex object.

Operator functions, not driven by the class type objects known as non-member functions, are kept as friends to the class. *friend* is the keyword used to declare such functions. A *friend* function to a class also has access to the private members of that class.

friend Complex operator + ( double d, Complex c)

{

Complex temp ;

temp . real = d + c. real ; // d is added into the real part of c

temp . imag = c. imag ;

return temp ;

}

You might have noticed that all the three overloaded + operator functions are accepting and returning variables by value. To make these functions better, we can also use references. So our first member + operators prototype can be rewritten as:

friend Complex operator + ( double d, Complex &c)

{

c. real = d + c. real ; // d is added into the real part of c

c. imag = c. imag ;

return c;

}

Now this operator function is accepting a complex number *Complex* by reference and returning a reference to the resultant complex number.

As discussed above, in case of assignment, the default assignment operator is used because we have not implemented (overloaded) our own assignment operator (=). But in case, we want to perform the following operation where the two operands are added and the resultant is assigned to one of them as:

c1 = c1 + c2;

There is one operator (+=) that can be used to do both the operations of addition and assignment instead of doing these operations separately within operator + and operator =. So we can overload this one operator (+=) here to make the code more efficient and reduce our work. Therefore, instead of writing:

c1 = c1 + c2;

We will write:

c1 += c2;

We will write our operator += as:

void Complex :: operator += ( Complex & c )

{

real += c. real ;

imag += c. imag ;

}

To use pre increment operator in your complex class, you can define it in following way:

Complex Complex :: operator ++ ()

{

Complex C;

C.real += 1 ;

C.imag += 1 ;

return C;

}

To overload insertion operator, we will define function as follow:

|  |
| --- |
| istream& operator>>( istream& in, Complex& c )  {  return in >> c.real >> c.imag ;  } |

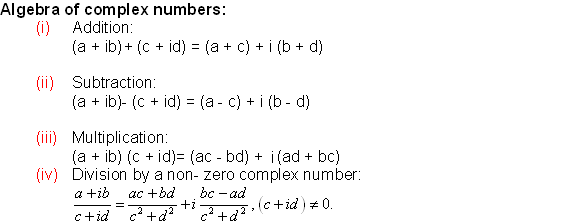
Make this function friend of complex class. Then in main you can write:

|  |
| --- |
| int main()  {  complex x;  cin>>c;  } |

# Practice Problem

1. Write a C++ program that can perform following operations on a complex class:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Operator To be Overloaded** | **Function Type** |
| Addition | - | Member |
| Subtraction | + | Member |
| Multiplication | \* | Member |
| Division | / | Non-member |
| Pre-Increment and post-decrement | ++, -- | Non-member |
| Overload Insertion and extraction operator | <<,>> | Non-member |



1. Write a C++ program that can perform following operations on a matrix class:

class Matrix

{

int arr[2][2];

};

* Using parameterized constructor, initialize matrix class object in the main().
* Overload the insertion operator and ask the user to enter the second array.
* Overload “+” operator for matrix addition.
* Overload “!” to find the determinant of matrix.
* Overload extraction operator, to display array on the console.