Experiment 4
Operator Overloading

Objectives

What operator overloading is and how it simplifies programming.
To overload operators for user-defined classes.
To overload unary and binary operators.
To convert objects from one class to another class.

Prelab Activities

Programming Output

For each of the given program segments, read the code and write the output in the space provided below each program. [Note: Do not execute these programs on a computer.]

For Programming Output Exercises 1–2, use the following class definition.

```
// Array.h
// Simple class Array (for integers)
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

// class Array definition
class Array
{
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );

public:
   Array( int = 10 ); // default constructor
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size
   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

   // determine if two arrays are not equal and
   // return true, otherwise return false (uses operator==)
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right );

   } // end function operator!=

   int &operator[]( int ); // subscript operator
   const int &operator[]( int ) const; // subscript operator
   static int getArrayCount(); // return number of
   // arrays instantiated
private:
   int size; // size of array
   int *ptr; // pointer to first element of array
   static int arrayCount; // number of Arrays instantiated

}; // end class Array

#endif // ARRAY_H
```

```
// Array.cpp
// Member function definitions for class Array
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <new>
using namespace std;

#include "Array.h"

// initialize static data member at file scope
int Array::arrayCount = 0; // no objects yet

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ]; // create space for array
   ++arrayCount; // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0; // initialize array

} // end class Array constructor

// copy constructor for class Array
// must receive reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy ) : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for array
   ++arrayCount; // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy arayToCopy into object

} // end copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr; // reclaim space for array
   --arrayCount; // one fewer object

} // end class Array destructor

// get size of array
int Array::getSize() const
{
   return size;

} // end function getSize

// overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
```

```
{
   if ( &right != this ) { // check for self-assignment

      // for arrays of different sizes, deallocate original
      // left side array, then allocate new left side array
      if ( size != right.size ) {
         delete [] ptr; // reclaim space
         size = right.size; // resize this object
         ptr = new int[ size ]; // create space for array copy

      } // end if

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ]; // copy array into object

   } // end if

   return *this; // enables x = y = z;

} // end function operator=

// determine if two arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different sizes

   for ( int i = 0; i < size; i++ )

      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true; // arrays are equal

} // end function operator==

// overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   if ( subscript < 0 || subscript >= size ) {
      cout << "\nError: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 ); // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // reference return

} // end function operator[]

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   if ( subscript < 0 || subscript >= size ) {
      cout << "\nError: Subscript " << subscript
```

```
         << " out of range" << endl;

      exit( 1 ); // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // const reference return

} // end function operator[]

// return number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount()
{
   return arrayCount;

} // end function getArrayCount

// overloaded input operator for class Array;
// inputs values for entire array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input; // enables cin >> x >> y;

} // end function operator>>

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;

   } // end for

   if ( i % 4 != 0 )
      output << endl;

   return output;

} // end function operator<<
```

1. What is output by the following code? Use the definition of class Array provided above.

```
#include "Array.h"

int main()
{
   cout << "# of arrays instantiated = "
       << Array::getArrayCount()  << '\n';

   Array integers1( 4 );
   Array integers2;

```

```cpp
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n";

   Array integers3( 8 ), *intptr = &integers2;

   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n\n";
} // end main
```

2. What is the output of the following program? Use the Array class shown above.

```cpp
#include "Array.h"

int main()
{
   Array integers1( 4 );
   Array integers2( 4 );

   if ( integers1 != integers2 )
      cout << "Hello";
   else
      cout << "Goodbye" << endl;
} // end main
```

Lab Exercises

Lab Exercise 1 — Vector Implementation

The problem is divided into six parts:

1. Lab Objectives
2. Description of the Problem
3. UML Diagram
4. Sample Output
5. Test Code Template
6. Problem-Solving Tips

The test program template represents a complete working C++ program test application. Read the problem description and examine the sample output; then study the template code. Using the problem-solving tips as a guide, write your C++ code. Compile and execute the program. Compare your output with the sample output provided.

Lab Objectives

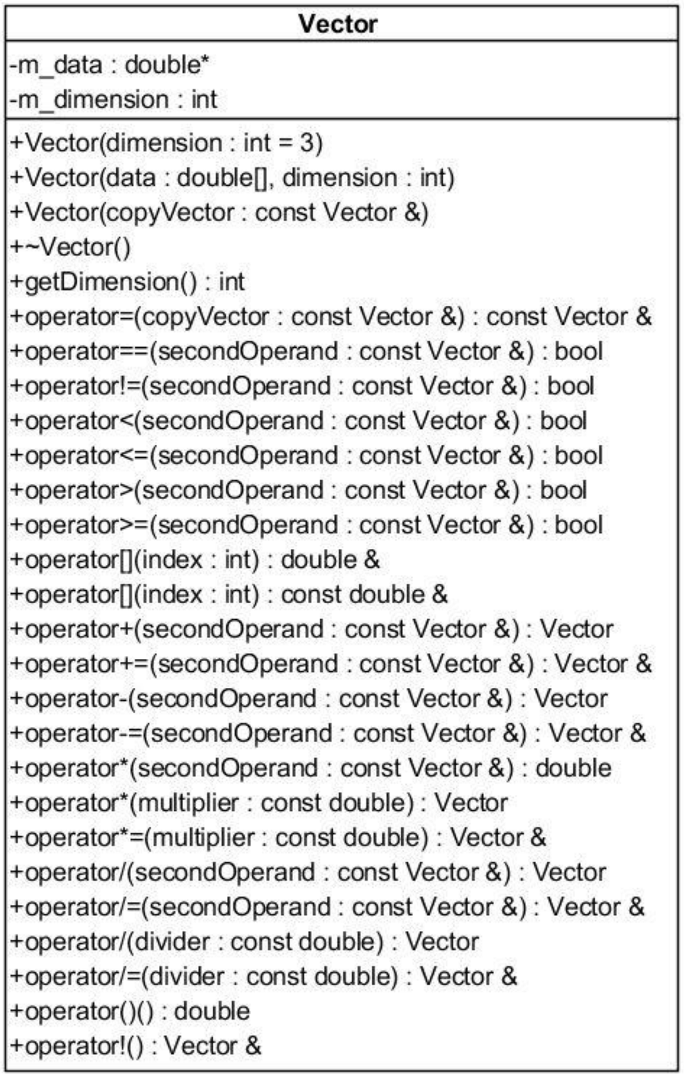
In this lab, you will practice:

- Overloading the + operator to allow String objects to be concatenated.
- Writing function prototypes for overloaded operators.
- Using overloaded operators.

Description of the Problem

Implement a Vector class that represents a mathematical vector. Look at the UML Diagram and implement required functionalities. Required functions is explained in the UML Diagram section. You are given a non-complete test code. Complete the test code according to the sample output. Test your implementation.

UML Diagram



Vector

-m_data : double*
-m_dimension : int

+Vector(dimension : int = 3)
+Vector(data : double[], dimension : int)
+Vector(copyVector : const Vector &)
+~Vector()
+getDimension() : int
+operator=(copyVector : const Vector &) : const Vector &
+operator==(secondOperand : const Vector &) : bool
+operator!=(secondOperand : const Vector &) : bool
+operator<(secondOperand : const Vector &) : bool
+operator<=(secondOperand : const Vector &) : bool
+operator>(secondOperand : const Vector &) : bool
+operator>=(secondOperand : const Vector &) : bool
+operator[](index : int) : double &
+operator[](index : int) : const double &
+operator+(secondOperand : const Vector &) : Vector
+operator+=(secondOperand : const Vector &) : Vector &
+operator-(secondOperand : const Vector &) : Vector
+operator-=(secondOperand : const Vector &) : Vector &
+operator*(secondOperand : const Vector &) : double
+operator*(multiplier : const double) : Vector
+operator*=(multiplier : const double) : Vector &
+operator/(secondOperand : const Vector &) : Vector
+operator/=(secondOperand : const Vector &) : Vector &
+operator/(divider : const double) : Vector
+operator/=(divider : const double) : Vector &
+operator()() : double
+operator!() : Vector &

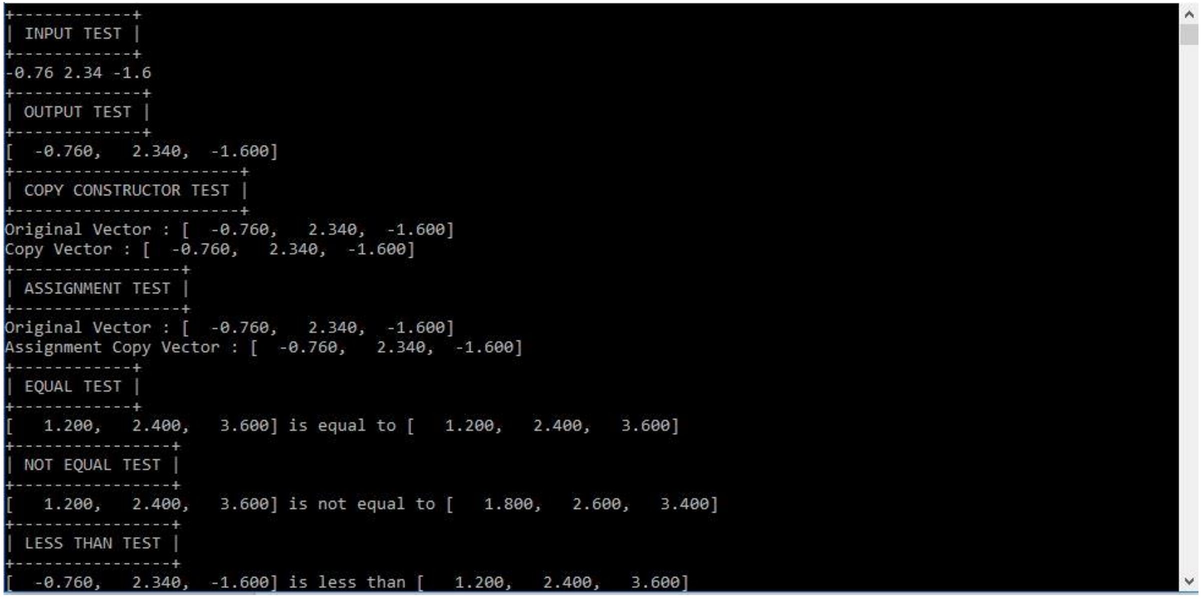
Members:

- m_dimension: size of the vector
- m_data: a double array to keep the raw data

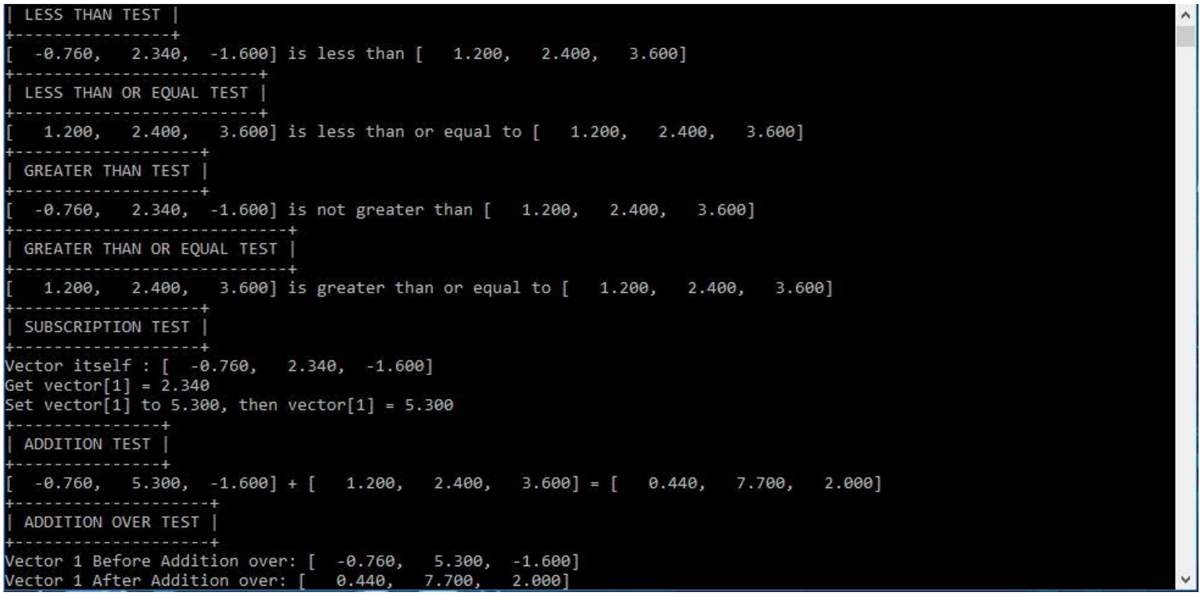
Functions:

- Vector(dimension = 3): Default parameter constructor
- Vector(data,dimension): Overloaded constructor with a double array and dimension parameter
- Vector(copyVector): Copy constructor
- ~Vector(): Destructor. You have to free the data array to handle the memory leak
- getDimension() : Returns the dimension of the vector
- Equal operator: Return true, if the vectors are equal
- Not equal operator: Returns true if the vectors are not equal
- Less than, greater than operators decides according to the magnitude of the vectors.
- Assign operator: Copies the vector given in the argument
- Subscription operator: Returns the vector element according to the given index

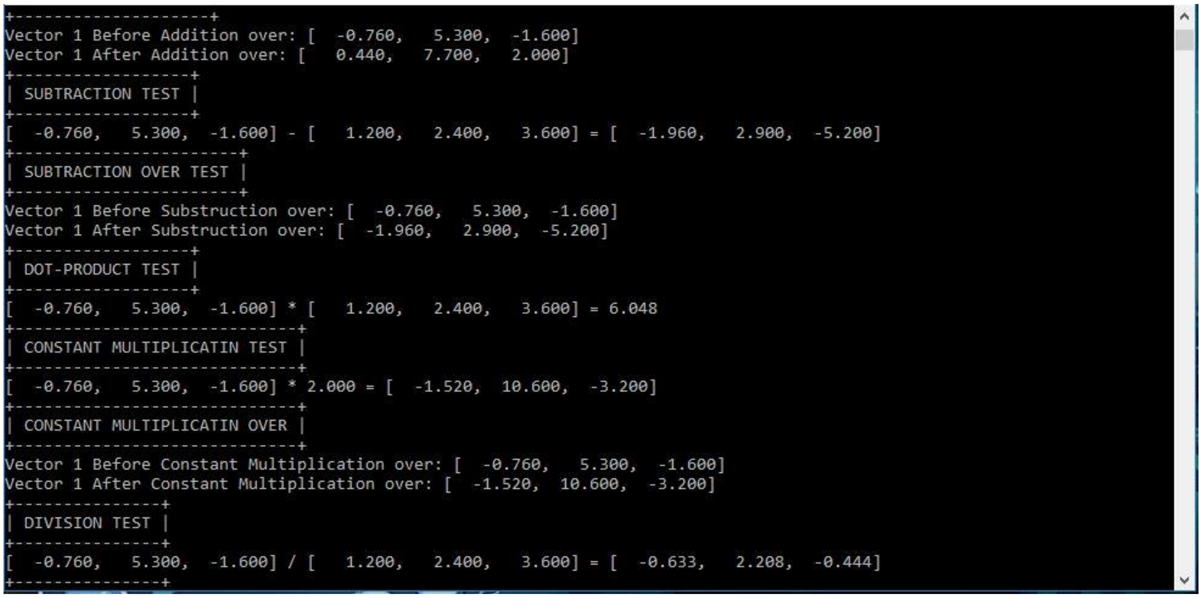
Sample Output



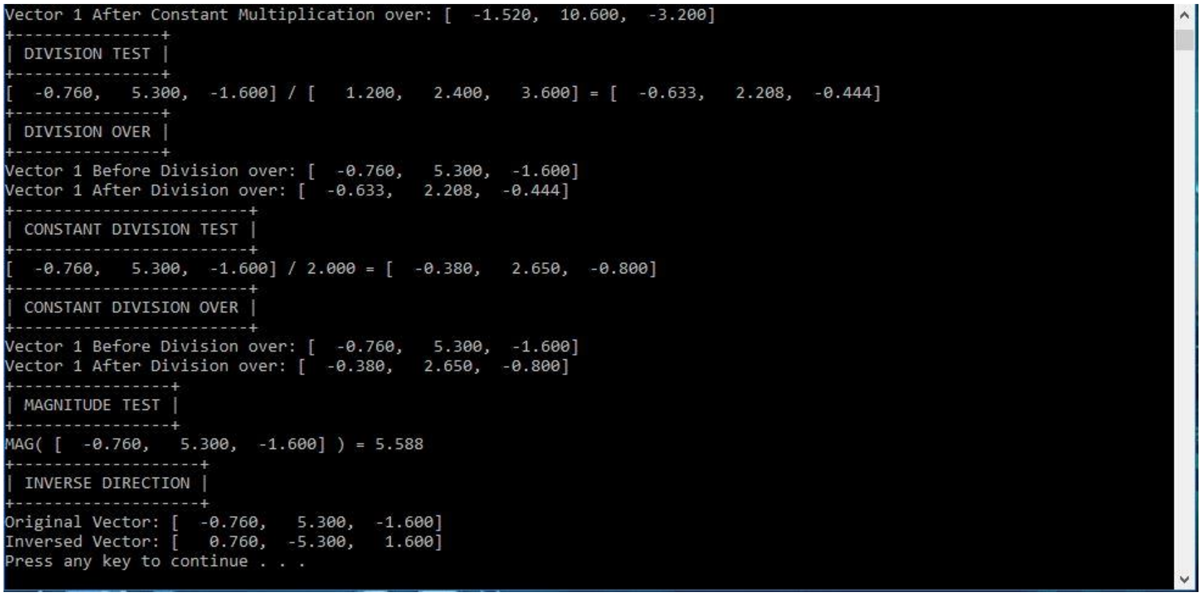
```
+------------+
| INPUT TEST |
+------------+
-0.76 2.34 -1.6
+-------------+
| OUTPUT TEST |
+-------------+
[  -0.760,   2.340,  -1.600]
+----------------------+
| COPY CONSTRUCTOR TEST |
+----------------------+
Original Vector : [  -0.760,   2.340,  -1.600]
Copy Vector : [  -0.760,   2.340,  -1.600]
+-----------------+
| ASSIGNMENT TEST |
+-----------------+
Original Vector : [  -0.760,   2.340,  -1.600]
Assignment Copy Vector : [  -0.760,   2.340,  -1.600]
+------------+
| EQUAL TEST |
+------------+
[   1.200,   2.400,   3.600] is equal to [   1.200,   2.400,   3.600]
+----------------+
| NOT EQUAL TEST |
+----------------+
[   1.200,   2.400,   3.600] is not equal to [   1.800,   2.600,   3.400]
+----------------+
| LESS THAN TEST |
+----------------+
[  -0.760,   2.340,  -1.600] is less than [   1.200,   2.400,   3.600]
```



```
| LESS THAN TEST |
+----------------+
[  -0.760,   2.340,  -1.600] is less than [   1.200,   2.400,   3.600]
+-------------------------+
| LESS THAN OR EQUAL TEST |
+-------------------------+
[   1.200,   2.400,   3.600] is less than or equal to [   1.200,   2.400,   3.600]
+-------------------+
| GREATER THAN TEST |
+-------------------+
[  -0.760,   2.340,  -1.600] is not greater than [   1.200,   2.400,   3.600]
+----------------------------+
| GREATER THAN OR EQUAL TEST |
+----------------------------+
[   1.200,   2.400,   3.600] is greater than or equal to [   1.200,   2.400,   3.600]
+-------------------+
| SUBSCRIPTION TEST |
+-------------------+
Vector itself : [  -0.760,   2.340,  -1.600]
Get vector[1] = 2.340
Set vector[1] to 5.300, then vector[1] = 5.300
+---------------+
| ADDITION TEST |
+---------------+
[  -0.760,   5.300,  -1.600] + [   1.200,   2.400,   3.600] = [   0.440,   7.700,   2.000]
+--------------------+
| ADDITION OVER TEST |
+--------------------+
Vector 1 Before Addition over: [  -0.760,   5.300,  -1.600]
Vector 1 After Addition over: [   0.440,   7.700,   2.000]
```



```
+-------------------+
Vector 1 Before Addition over: [  -0.760,   5.300,  -1.600]
Vector 1 After Addition over: [   0.440,   7.700,   2.000]
+-----------------+
| SUBTRACTION TEST |
+-----------------+
[  -0.760,   5.300,  -1.600] - [   1.200,   2.400,   3.600] = [  -1.960,   2.900,  -5.200]
+----------------------+
| SUBTRACTION OVER TEST |
+----------------------+
Vector 1 Before Substruction over: [  -0.760,   5.300,  -1.600]
Vector 1 After Substruction over: [  -1.960,   2.900,  -5.200]
+-----------------+
| DOT-PRODUCT TEST |
+-----------------+
[  -0.760,   5.300,  -1.600] * [   1.200,   2.400,   3.600] = 6.048
+----------------------------+
| CONSTANT MULTIPLICATIN TEST |
+----------------------------+
[  -0.760,   5.300,  -1.600] * 2.000 = [  -1.520,  10.600,  -3.200]
+----------------------------+
| CONSTANT MULTIPLICATIN OVER |
+----------------------------+
Vector 1 Before Constant Multiplication over: [  -0.760,   5.300,  -1.600]
Vector 1 After Constant Multiplication over: [  -1.520,  10.600,  -3.200]
+--------------+
| DIVISION TEST |
+--------------+
[  -0.760,   5.300,  -1.600] / [   1.200,   2.400,   3.600] = [  -0.633,   2.208,  -0.444]
+--------------+
```



```
Vector 1 After Constant Multiplication over: [  -1.520,  10.600,  -3.200]
+--------------+
| DIVISION TEST |
+--------------+
[  -0.760,   5.300,  -1.600] / [   1.200,   2.400,   3.600] = [  -0.633,   2.208,  -0.444]
+--------------+
| DIVISION OVER |
+--------------+
Vector 1 Before Division over: [  -0.760,   5.300,  -1.600]
Vector 1 After Division over: [  -0.633,   2.208,  -0.444]
+-----------------------+
| CONSTANT DIVISION TEST |
+-----------------------+
[  -0.760,   5.300,  -1.600] / 2.000 = [  -0.380,   2.650,  -0.800]
+-----------------------+
| CONSTANT DIVISION OVER |
+-----------------------+
Vector 1 Before Division over: [  -0.760,   5.300,  -1.600]
Vector 1 After Division over: [  -0.380,   2.650,  -0.800]
+---------------+
| MAGNITUDE TEST |
+---------------+
MAG( [  -0.760,   5.300,  -1.600] ) = 5.588
+------------------+
| INVERSE DIRECTION |
+------------------+
Original Vector: [  -0.760,   5.300,  -1.600]
Inversed Vector: [   0.760,  -5.300,   1.600]
Press any key to continue . . .
```

Test Code Template

```
/******************************************
******************************************
* IDE : Visual Studio 2015              *
* Author : Cihan UYANIK                 *
* Experiment 4: Operator Overloading    *
*****************************************/
#include "Vector.h"

void TEST_Input(Vector& vector)
{
	cout << "+------------+" << endl
		<< "| INPUT TEST |" << endl
		<< "+------------+" << endl;
	cin >> vector;
}
```

```
void TEST_Output(Vector& vector)
{
       cout << "+-------------+" << endl
              << "| OUTPUT TEST |" << endl
              << "+-------------+" << endl;
       cout << vector << endl;
}

void TEST_CopyConstructor(Vector& vector)
{
       cout <<"+----------------------+" << endl
              << "| COPY CONSTRUCTOR TEST |" << endl
              << "+----------------------+" << endl;

       Vector copy_vector(vector);

       cout << "Original Vector : " <<vector<< endl<< "Copy Vector : "<<copy_vector
<< endl;
}

void TEST_Assignment(Vector& vector)
{

       cout<< "+-----------------+" << endl
              << "| ASSIGNMENT TEST |" << endl
              << "+-----------------+" << endl;

       Vector copy_vector;

       copy_vector = vector;

       cout << "Original Vector : " << vector << endl << "Assignment Copy Vector : "
<< copy_vector << endl;
}

void TEST_Equal(Vector& vector1, Vector& vector2)
{
       cout <<"+------------+" << endl
              << "| EQUAL TEST |" << endl
              << "+------------+" << endl;

       if (vector1 == vector2)
       {
              cout << vector1 << " is equal to " << vector2 << endl;
       }
       else
       {
              cout << vector1 << " is not equal to " << vector2 << endl;
       }
}

void TEST_Not_Equal(Vector& vector1, Vector& vector2)
{
// Implement the function
}
```

```
void TEST_LESS_THAN(Vector& vector1, Vector& vector2)
{
       cout << "+----------------+" << endl
              << "| LESS THAN TEST |" << endl
              << "+----------------+" << endl;

       if (vector1 < vector2)
       {
              cout << vector1 << " is less than " << vector2 << endl;
       }
       else
       {
              cout << vector1 << " is not less than " << vector2 << endl;
       }
}

void TEST_LESS_THAN_OR_EQUAL(Vector& vector1, Vector& vector2)
{
// Implement the function
}

void TEST_GREATER_THAN(Vector& vector1, Vector& vector2)
{
// Implement the function
}

void TEST_GREATER_THAN_OR_EQUAL(Vector& vector1, Vector& vector2)
{
// Implement the function
}

void TEST_Subscription(Vector& vector, int i, double newValue)
{
       cout <<"+-------------------+" << endl
              << "| SUBSCRIPTION TEST |" << endl
              << "+-------------------+" << endl;

       cout << "Vector itself : " << vector << endl;
       cout << "Get vector[" << i << "] = " << vector[i] << endl;
       vector[i] = newValue;
       cout << "Set vector[" << i << "] to "<<newValue<<", then vector[" << i << "] =
" << vector[i] << endl;
}

void TEST_Addition(Vector& vector1, Vector& vector2)
{
       cout <<"+---------------+" << endl
              << "| ADDITION TEST |" << endl
              << "+---------------+" << endl;

       Vector result = vector1 + vector2;

       cout << vector1 << " + " << vector2 << " = " << result << endl;
}
```

```
void TEST_AdditionOver(Vector vector1, Vector vector2)
{
       cout << "+--------------------+" << endl
              << "| ADDITION OVER TEST |" << endl
              << "+--------------------+" << endl;
       cout << "Vector 1 Before Addition over: " << vector1 << endl;
       vector1 += vector2;
       cout << "Vector 1 After Addition over: " << vector1 << endl;
}

void TEST_Substraction(Vector& vector1, Vector& vector2)
{
// Implement the function}

void TEST_SubstractionOver(Vector vector1, Vector vector2)
{
// Implement the function
}

void TEST_DotProduct(Vector& vector1, Vector& vector2)
{
// Implement the function}

void TEST_Constant_Multiplication(Vector& vector1,  double constant_value)
{
// Implement the function}

void TEST_Constant_MultiplicationOver(Vector vector, double constant_value)
{
       cout << "+------------------------------+" << endl
              << "| CONSTANT MULTIPLICATIN OVER |" << endl
              << "+------------------------------+" << endl;
       cout << "Vector 1 Before Constant Multiplication over: " << vector << endl;
       vector *= constant_value;
       cout << "Vector 1 After Constant Multiplication over: " << vector << endl;
}

void TEST_Division(Vector& vector1, Vector& vector2)
{
// Implement the function}

void TEST_DivisionOver(Vector vector1, Vector vector2)
{
// Implement the function}

void TEST_Constant_Division(Vector& vector1, double constant_value)
{
// Implement the function
}

void TEST_Constant_DivisionOver(Vector vector, double constant_value)
{
// Implement the function
}
```

```
void TEST_Magnitude(Vector& vector1)
{
       cout <<"+----------------+" << endl
              << "| MAGNITUDE TEST |" << endl
              << "+----------------+" << endl;
       double result = vector1();
       cout << "MAG( " << vector1 << " ) = " << result << endl;
}

void TEST_InverseDirection(Vector vector)
{
       cout << "+-------------------+" << endl
              << "| INVERSE DIRECTION |" << endl
              << "+-------------------+" << endl;
       cout << "Original Vector: " << vector << endl;
       cout << "Inversed Vector: " << !vector << endl;
}

int main()
{
       double firstTestData[] {1.2, 2.4, 3.6};
       double secondTestData[] {1.8, 2.6, 3.4};
       // Implement the function
       Vector v1(3);
       Vector v2(firstTestData, 3);
       Vector v3(firstTestData, 3);
       Vector v4(secondTestData, 3);
       TEST_Input(v1);
       TEST_Output(v1);
       TEST_CopyConstructor(v1);
       TEST_Assignment(v1);
       TEST_Equal(v2, v3);
       TEST_Not_Equal(v3, v4);
       TEST_LESS_THAN(v1, v2);
       TEST_LESS_THAN_OR_EQUAL(v2, v3);
       TEST_GREATER_THAN(v1, v2);
       TEST_GREATER_THAN_OR_EQUAL(v2, v3);
       TEST_Subscription(v1, 1, 5.3);
       TEST_Addition(v1, v2);
       TEST_AdditionOver(v1, v2);
       TEST_Substraction(v1, v2);
       TEST_SubstractionOver(v1, v2);
       TEST_DotProduct(v1, v2);
       TEST_Constant_Multiplication(v1, 2);
       TEST_Constant_MultiplicationOver(v1, 2);
       TEST_Division(v1, v2);
       TEST_DivisionOver(v1, v2);
       TEST_Constant_Division(v1, 2);
       TEST_Constant_DivisionOver(v1, 2);
       TEST_Magnitude(v1);
       TEST_InverseDirection(v1);
       return 0;
}
```

Problem-Solving Tips

1- Use given UML Diagram and Test Code Template.