Şafak AKINCI
152120141003

22.12.2016

Experiment 6
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

1-
```
#of arrays instantiated = 0
#of arrays instantiated = 2
#of arrays instantiated = 4
```

2-
```
Goodbye
```

Lab Exercises

Lab Exercise 1 – VectorImplementation

```
/*****************************************
 * VectorImplementation.h                *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/
```

```
#include <iostream>             //To use standart input and output functions (cin and cout).

//Vector class that represents a mathematical vector.
class Vector
{
public:
    //Default parameter constructor.
    Vector(int dimension = 3);

    //Overloaded constructor with a double array and dimension parameter.
    Vector(double data[], int dimension);

    //Copy constructor, it takes one constant parameter of its kind as reference.
    Vector(const Vector& copyVector);

    //Destructor. Free the data array to handle the memory leak.
    ~Vector();

    //Returns the dimension of the vector.
    int getDimension();

    //Copies the given vector in the argument.
    //Returns this object (vector) as constant (const Vector&).
```

```
const Vector& operator= (const Vector& copyVector);

//Take inputs for vector. This function is class' friend, so function reach all members of this class.
friend std::istream& operator>>(std::istream&, Vector&);

//Print vector's data. This function is class' friend, so function reach all members of this class.
friend std::ostream& operator<<(std::ostream&, Vector&);

//Return true, if the vectors are equal.
bool operator== (const Vector& secondOperand);

//Returns true, if the vectors are not equal.              REVERSE FUNCTION OF ==.
bool operator!= (const Vector& secondOperand);

//Decides according to the magnitude of the vectors.
bool operator< (const Vector& secondOperand);

//Decides according to the magnitude of the vectors.       REVERSE FUNCTION OF <
bool operator>= (const Vector& secondOperand);

//Decides according to the magnitude of the vectors.
bool operator> (const Vector& secondOperand);

//Decides according to the magnitude of the vectors.       REVERSE FUNCTION OF >
bool operator<= (const Vector& secondOperand);

//Returns the vector element according to the given index and it works for non-const objects.
double& operator[] (int index);

//Returns the vector element according to the given index and it works for const objects.
const double& operator[](int index) const;

//Adds two vectors and returns a Vector object.
Vector operator+ (const Vector& secondOperand);

//Adds secondOperand's data to object, and returns this object. (Vector&)
Vector& operator+= (const Vector& secondOperand);

//Subtracts two vectors and returns a Vector object.
Vector operator- (const Vector& secondOperand);

//Subtracts secondOperand's data from object, and returns this object. (Vector&)
Vector& operator-= (const Vector& secondOperand);

//Dot Product. Multiplies each element of object with secondOperand's element and adds them.
double operator* (const Vector& secondOperand);
```

```
    //Multiples object's data with multiplier, and returns a Vector object.
    Vector operator* (const double multiplier);

    //Multiplies secondOperand's data with object's data, and returns this object. (Vector&)
    Vector& operator*= (const double multiplier);

    //Divides object's data to secondOperand's data, and returns a Vector object.
    Vector operator/ (const Vector& secondOperand);

    //Divides object's data to secondOperand's data, and returns this object. (Vector&)
    Vector& operator/= (const Vector& secondOperand);

    //Divides object's data to divider, and returns a Vector object.
    Vector operator/ (const double divider);

    //Divides object's data to divider, and returns this object. (Vector&)
    Vector& operator/= (const double divider);

    //Calculates object's magnitude.
    double operator() ();

    //Multiply object with -1.
    Vector& operator! ();
private:
    double* m_data;             //A double array to keep the raw data.
    int m_dimension;            //Size of the vector.
};

/*****************************************
 * VectorImplementation.cpp              *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/

#include <iostream>                 //To use standart input and output functions like cin and cout.
#include <math.h>                   //To use pow() and sqrt() functions!
#include "VectorImplementation.h"   //To know function prototypes of Vector class.

//Use cin and cout functions under the std namespace.
//And also create an object from istream and ostream classes which are defined in std namespace.
using namespace std;

//Default parameter constructor.
//m_dimension is initialized by using member initializer :
```

```
Vector::Vector(int dimension):m_dimension(dimension)
{
    //Memory is allocated for object.
    m_data = new double [m_dimension];
}

//Overloaded constructor with a double array and dimension parameter.
Vector::Vector(double data[], int dimension):m_dimension(dimension)
{
    //Memory is allocated for object.
    m_data = new double [dimension];

    //data's elements are assigned to m_data's element of vector.
    for(int i=0; i<m_dimension; i++)
        m_data[i]= data[i];
}

//Copy constructor, it takes one constant parameter of its kind as reference.
Vector::Vector(const Vector& copyVector):m_dimension(copyVector.m_dimension)
{
    //m_data is a pointer the type of double, and each object should be its own array.
    m_data = new double[m_dimension];

    //copyVector's data is assigned to vector's data.
    for(int i=0; i<m_dimension; i++)
        m_data[i]= copyVector.m_data[i];

}

//Destructor function of Vector class.
//Free the data array to handle the memory leak.
Vector::~Vector()
{
    delete[] m_data;
}

//Returns the dimension of the vector.
int Vector::getDimension()
{
    return m_dimension;
}

//Takes inputs for vector. This function is class' friend, so function reach all members of this class.
istream& operator>>(istream& in, Vector& vec)
{
    for(int i=0; i<vec.m_dimension; i++)
        in>>vec[i];
```

```
    //Returns istream&
    return in;              //Enables cascading.
}

//Prints vector's data. This function is class' friend, so function reach all members of this class.
ostream& operator<<(ostream& os, Vector& vec)
{
    cout<<"[ ";
    for(int i=0; i<vec.m_dimension-1; i++)
        os<<"\t"<<vec.m_data[i]<<",";

    cout<<"\t"<<vec.m_data[vec.m_dimension-1]<<"]";

    //Returns ostream&
    return os;              //Enables cascading.
}

//Copies the given vector in the argument.
const Vector& Vector::operator= (const Vector& copyVector)
{
    //Avoids self assignment for vectors of different sizes, deallocate original left-side vector.
    if (&copyVector != this)
    {
        //then allocate new left-side vector.
        if(m_dimension != copyVector.m_dimension)
        {
            delete [] m_data;                               //release space
            m_dimension = copyVector.m_dimension;           //resize this object
            m_data = new double [m_dimension];              //create space for array copy
        }

        for(int i=0; i<m_dimension; i++)
            m_data[i] = copyVector.m_data[i];

    }//end if

    //Returns this object (vector) as constant (const Vector&).
    return *this;
}

//Return true, if the vectors are equal.
bool Vector::operator== (const Vector& secondOperand)
{
    if(m_dimension != secondOperand.m_dimension)
        return false;                               //Arrays' sizes are not equal!
```

```
    for(int i=0; i<m_dimension; i++)
        if(m_data[i]!= secondOperand.m_data[i])
            return false;                               //Arrays' contents are not equal!

    return true;                                        //Arrays are equal.
}

//Returns true if the vectors are not equal.
//REVERSE FUNCTION OF ==.
bool Vector::operator!= (const Vector& secondOperand)
{
    return !( (*this) == secondOperand);
    /*
     if(m_dimension != secondOperand.m_dimension)           //Arrray's sizes are not equal, they can not be equal.
     return true;

     for(int i=0; i<m_dimension; i++)
     if(m_data[i] == secondOperand.m_data[i])
     return false;

     return true;
     */
}

//Decides according to the magnitude of the vectors.
bool Vector::operator<(const Vector& secondOperand)
{
    double magnitudeFirstVector  = (*this)();

    //secondOperand is constant vector, operator() cannot be called for it.
    double magnitudeSecondVector = 0;
    for(int i=0; i<secondOperand.m_dimension; i++)
        magnitudeSecondVector += pow(secondOperand.m_data[i],2);

    magnitudeSecondVector = sqrt(magnitudeSecondVector);

    if(magnitudeFirstVector < magnitudeSecondVector)
        return true;

    return false;
}

//Decides according to the magnitude of the vectors.
//REVERSE FUNCTION OF <
bool Vector::operator>=(const Vector& secondOperand)
{
    return !( (*this)<secondOperand );
```

```
}

//Decides according to the magnitude of the vectors.
bool Vector::operator>(const Vector& secondOperand)
{
    double magnitudeFirstVector=0;
    for(int i=0; i<m_dimension; i++)
        magnitudeFirstVector += pow(m_data[i],2);

    magnitudeFirstVector = sqrt(magnitudeFirstVector);

    double magnitudeSecondVector=0;
    for(int i=0; i<secondOperand.m_dimension; i++)
        magnitudeSecondVector += pow(secondOperand.m_data[i],2);

    magnitudeSecondVector = sqrt(magnitudeSecondVector);

    if(magnitudeFirstVector > magnitudeSecondVector)
        return true;

    return false;
}

//Decides according to the magnitude of the vectors.
//REVERSE FUNCTION OF >
bool Vector::operator<=(const Vector& secondOperand)
{
    return !( (*this)>secondOperand );
}

//Returns the vector element according to the given index and it works for non-const objects.
double& Vector::operator[] (int index)
{
    if(index>=0 ||index < m_dimension)
        return m_data[index];

    cout<<"You tried to reach an element that is out of the vector!\n";
    return m_data[0];
}

//Returns the vector element according to the given index and it works for const objects.
const double& Vector::operator[](int index) const
{
    if(index>=0 || index < m_dimension)
        return m_data[index];

    cout<<"You tried to reach an element that is out of the vector!\n";
```

```
    return m_data[0];
}

//Adds two vectors and returns a Vector object.
Vector Vector::operator+ (const Vector& secondOperand)
{
    Vector result;

    for(int i=0; i<m_dimension; i++)
        result.m_data[i]= m_data[i]+ secondOperand.m_data[i];

    return result;
}

//Adds secondOperand's data to object, and returns this object. (Vector&)
Vector& Vector::operator+= (const Vector& secondOperand)
{
    for(int i=0; i<m_dimension; i++)
        m_data[i] += secondOperand.m_data[i];

    return (*this);
}

//Subtracts two vectors and returns a Vector object.
Vector Vector::operator- (const Vector& secondOperand)
{
    Vector result;

    for(int i=0; i<m_dimension; i++)
        result.m_data[i] = m_data[i]- secondOperand.m_data[i];

    return result;
}

//Subtracts secondOperand's data from object, and returns this object. (Vector&)
Vector& Vector::operator-= (const Vector& secondOperand)
{
    for(int i=0; i<m_dimension; i++)
        m_data[i] -= secondOperand.m_data[i];

    return (*this);
}

//Dot Product.Multiplies each element of object with secondOperand's element and adds them.
double Vector::operator* (const Vector& secondOperand)
{
    double result=0;
```

```
    for(int i=0; i<m_dimension; i++)
        result += m_data[i] * secondOperand.m_data[i];

    return result;
}

//Multiples object's data with multiplier, and returns a Vector object.
Vector Vector::operator* (const double multiplier)
{
    Vector result;

    for(int i=0; i<m_dimension; i++)
        result.m_data[i]= m_data[i]* multiplier;

    return result;
}

//Multiplies secondOperand's data with object's data, and returns this object. (Vector&)
Vector& Vector::operator*= (const double multiplier)
{
    for(int i=0; i<m_dimension; i++)
        m_data[i]*= multiplier;

    return (*this);
}

//Divides object's data to secondOperand's data, and returns a Vector object.
Vector Vector::operator/ (const Vector& secondOperand)
{
    Vector result;

    for(int i=0; i<m_dimension; i++)
        result.m_data[i]= m_data[i]/ secondOperand.m_data[i];

    return result;
}

//Divides object's data to secondOperand's data, and returns this object. (Vector&)
Vector& Vector::operator/= (const Vector& secondOperand)
{
    for(int i=0; i<m_dimension; i++)
        m_data[i] /= secondOperand.m_data[i];

    return (*this);
}
```

```cpp
//Divides object's data to divider, and returns a Vector object.
Vector Vector::operator/ (const double divider)
{
    Vector result;

    for(int i=0; i<m_dimension; i++)
        result.m_data[i]= m_data[i]/ divider;

    return result;
}

//Divides object's data to divider, and returns this object. (Vector&)
Vector& Vector::operator/= (const double divider)
{
    for(int i=0; i<m_dimension; i++)
        m_data[i] /= divider;

    return (*this);
}

//Calculates object's magnitude.
double Vector::operator() ()
{
    double result = 0;

    for(int i=0; i<m_dimension; i++)
        result += pow(m_data[i],2);

    return sqrt(result);
}

//Multiply object with -1.
Vector& Vector::operator! ()
{
    return  (*this)*=-1;
}

/*****************************************
 * VectorImplementationTestMain.cpp      *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/

#include <iostream>                 //To use standart input and output functions like cin and cout.
#include "VectorImplementation.h"   //To know function prototypes of Vector class.
```

```cpp
using namespace std;                //Use cin and cout functions under the std namespace.


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

    //Takes inputs and they will be assigned to vector's m_data.
    cin>>vector;
}

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

    //Prints vector's m_data to the console.
    cout<<vector<<endl;
}

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

    //copy_vector is created as vector's copy. (Same dimension, Same data.)
    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;

    //copy_vector is created, default constructor is called for it. (m_dimension = 3)
    Vector copy_vector;

    //vector's members are assigned to 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;

    //Checks vectors are equal.
    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)
{
    cout<< "+----------------+" << endl
    << "| NOT EQUAL TEST |" << endl
    << "+----------------+" << endl;

    //Checks vector are not equal.
    if(vector1 != vector2)
        cout << vector1 << " is not equal to " << vector2 << endl;
    else
        cout << vector1 << " is equal to " << vector2 << endl;

}

void TEST_LESS_THAN(Vector& vector1, Vector& vector2)
{

    cout<< "+----------------+" << endl
    << "| LESS THAN TEST |" << endl
    << "+----------------+" << endl;

    //Checks the left-hand side vectors is less than the other or not.
    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)
{
```

```
    cout<< "+-----------------------+" << endl
    << "| LESS THAN OR EQUAL TEST |" << endl
    << "+-----------------------+" << endl;

    //Checks the left-hand side vectors is less than or equal to the other or not.
    if (vector1 <= vector2)
        cout << vector1 << " is less than or equal to " << vector2 << endl;
    else
        cout << vector1 << " is not less than or equal to" << vector2 << endl;

}

void TEST_GREATER_THAN(Vector& vector1, Vector& vector2)
{
    cout<< "+--------------------+" << endl
    << "| GREATER THAN  TEST |" << endl
    << "+--------------------+" << endl;

    //Checks the left-hand side vectors is greater than the other or not.
    if (vector1 > vector2)
        cout << vector1 << " is greater than " << vector2 << endl;
    else
        cout << vector1 << " is not greater than " << vector2 << endl;
}

void TEST_GREATER_THAN_OR_EQUAL(Vector& vector1, Vector& vector2)
{
    cout<< "+--------------------------+" << endl
    << "| GREATER THAN OR EQUAL TEST |" << endl
    << "+--------------------------+" << endl;

    //Checks the left-hand side vectors is greater than or equal than the other or not.
    if (vector1 >= vector2)
        cout << vector1 << " is greater than or equal to " << vector2 << endl;
    else
        cout << vector1 << " is not greater than or equal to " << vector2 << endl;
}

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;
```

```
    //newValue is assigned to vector's i th element.
    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;

    //Addition operator adds given vectors, and returns final vector. Returned vector is assigned to result the type
of vector.
    Vector result = vector1 + vector2;
    cout << vector1 << " + " << vector2 << " = " << result << endl;
}

void TEST_AdditionOver(Vector vector1, Vector vector2)
{
    cout<< "+--------------------+" << endl
    << "| ADDITION OVER TEST |" << endl
    << "+--------------------+" << endl;
    //Addition Over operator adds vector2 to vector1, and returns vector1.
    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)
{
    cout<< "+--------------------+" << endl
    << "|  SUBTRACTION TEST  |" << endl
    << "+--------------------+" << endl;

    //Subtraction operator subtracts given vectors, and returns final vector. Returned vector is assigned to result
the type of vector.
    Vector result = vector1 - vector2;
    cout << vector1 << " - " << vector2 << " = " << result << endl;
}

void TEST_SubstractionOver(Vector vector1, Vector vector2)
{
    cout<< "+----------------------+" << endl
    << "| SUBTRACTION OVER TEST |" << endl
    << "+----------------------+" << endl;
    //Subtraction Over operator subtracts vector2 from vector1, and returns vector1.
    cout << "Vector 1 Before Subtraction over: " << vector1 << endl; vector1 -= vector2;
    cout << "Vector 1 After Subtraction over: " << vector1 << endl;
```

```
}

void TEST_DotProduct(Vector& vector1, Vector& vector2)
{
    cout<< "+------------------+" << endl
    << "| DOT PRODUCT TEST |" << endl
    << "+------------------+" << endl;

    //Each element of vector1 and vector2 will multiply each other and add, the final result is called DOT PRODUCT.
    cout << vector1 << " - " << vector2 << " = " << vector1*vector2 <<endl;
}

void TEST_Constant_Multiplication(Vector& vector1, double constant_value)
{
    cout<< "+-----------------------------+" << endl
    << "| CONSTANT MULTIPLICATION TEST |" << endl
    << "+-----------------------------+" << endl;

    //vector1's elements will multiply with constant_value and this operator returns a vector, it is assigned to
result (Vector).
    Vector result = vector1*constant_value;
    cout << vector1 << " * " << constant_value << " = " << result << endl;
}

void TEST_Constant_MultiplicationOver(Vector vector, double constant_value)
{
    cout<< "+-----------------------------+" << endl
    << "| CONSTANT MULTIPLICATIN OVER |" << endl
    << "+-----------------------------+" << endl;

    cout << "Vector Before Constant Multiplication over: " << vector << endl;
    //Multiplication Over operator multiply vector with constant_value and operator returns this vector.
    vector *= constant_value;
    cout << "Vector After Constant Multiplication over: " << vector << endl;
}

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

    //vector1's data will divide to vector2's data and Division operator returns a vector, returned vector will
assign to result (Vector).
    Vector result = vector1/vector2;
    cout << vector1 << " / " << vector2 << " = " << result << endl;
}
```

```
void TEST_DivisionOver(Vector vector1, Vector vector2)
{
    cout<< "+--------------------+" << endl
    << "| DIVISION OVER TEST |" << endl
    << "+--------------------+" << endl;

    cout << "Vector 1 Before Division over: " << vector1 << endl;
    //Division Over operator divides vector1 to vector2 and operator returns this vector.
    vector1 /= vector2;
    cout << "Vector 1 After Division over: " << vector1 << endl;
}

void TEST_Constant_Division(Vector& vector1, double constant_value)
{
    cout<< "+-----------------------+" << endl
    << "| CONSTANT DIVISION TEST |" << endl
    << "+-----------------------+" << endl;

    //Division operator divides vector to constant_value and operator returns a vector, returned vector will assign
to result (Vector).
    Vector result = vector1 / constant_value;
    cout << vector1 << " / " << constant_value << " = " << result << endl;
}

void TEST_Constant_DivisionOver(Vector vector, double constant_value)
{
    cout<< "+----------------------------+" << endl
    << "| CONSTANT DIVISION OVER TEST |" << endl
    << "+----------------------------+" << endl;

    cout << "Vector Before Constant Division over: " << vector << endl;

    //Division Over operator divides vector1 to constant_value and operator returns this vector.
    vector /= constant_value;
    cout << "Vector After Constant Division over: " << vector << endl;

}

void TEST_Magnitude(Vector& vector1)
{
    cout<< "+----------------+" << endl
    << "| MAGNITUDE TEST |" << endl
    << "+----------------+" << endl;
    //operator() will calculate the vector1's magnitude, and operator returns the result.
    double result = vector1();
```

```
    cout << "MAG( " << vector1 << " ) = " << result << endl;
}

void TEST_InverseDirection(Vector vector)
{
    cout<< "+-------------------+" << endl
    << "| INVERSE DIRECTION |" << endl
    << "+-------------------+" << endl;

    //operator! will multiply the vector with -1.
    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};

    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;
}

/*****************************************
 * CustomRectangle.h                     *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/

#include <iostream>                 //To use standart input and output functions like cin and cout.

//Use cin and cout functions under the std namespace.
//And also create an object from istream and ostream classes which are defined in std namespace.
using namespace std;

class CustomRectangle
{
    int m_width;                //mine_width
    int m_height;               //mine_height

public:
    //Constructor Function initializes private member values called m_width and m_height.
    CustomRectangle (int width =0, int height =0);

    //Destructor Function.
    ~CustomRectangle ();

    //Returns object's width (m_width)
    int getWidth ();
    //Returns object's height (m_height)
    int getHeight ();

    //Adds other's size with object's size and returns a CustomRectangle object.
    CustomRectangle operator+(const CustomRectangle& other);

    //Adds other's size to object's size, and returns this object. (CustomRectangle&)
    CustomRectangle& operator+=(const CustomRectangle& other);

    //Subtracts other's size from object's size and returns a CustomRectangle object.
    CustomRectangle operator-(const CustomRectangle& other);

    //Subtracts other's size from object's size, and returns this object. (CustomRectangle&)
    CustomRectangle& operator-=(const CustomRectangle& other);
```

```
    //Multiplies other's size with constantNumber and returns a CustomRectangle object.
    CustomRectangle operator*(int constantNumber);

    //Multiplies other's size with object's size and returns a CustomRectangle object.
    CustomRectangle operator*(const CustomRectangle& other);

    //Multiplies other's size with object's size, and returns this object. (CustomRectangle&)
    CustomRectangle& operator*=(const CustomRectangle& other);

    //Increment operator will increase the width and height by one. PREFIX FORM.
    CustomRectangle& operator++ ();

    //Decrement operator will decrease the width and height by one.
    CustomRectangle& operator-- ();

    //Prints rectangle's size. This function is class' friend, so function reach all members of this class.
    friend ostream& operator<<(ostream& out, const CustomRectangle& rectangle);

    //Takes inputs for rectangle. This function is class' friend, so function reach all members of this class.
    friend istream& operator>>(istream& in, CustomRectangle& rectangle);
};
```

```
/*****************************************
 * CustomRectangle.cpp                   *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/

#include "CustomRectangle.h"            //To know function prototypes of CustomRectangle class.
#include <iostream>                     //To use standart input and output functions (cin, cout).

//Constructor Function.
CustomRectangle::CustomRectangle (int width, int height):m_width(width),m_height(height)
{}

//Destructor Function.
CustomRectangle::~CustomRectangle()
{};

//Returns object's m_width.
int CustomRectangle::getWidth()
{
    return m_width;
```

```
}

//Returns object's m_height.
int CustomRectangle::getHeight()
{
    return m_height;
}

//Adds other's size with object's size and returns a CustomRectangle object called rect.
CustomRectangle CustomRectangle::operator+(const CustomRectangle& other)
{
    CustomRectangle rect;
    rect.m_width = m_width + other.m_width;
    rect.m_height = m_height + other.m_height;

    return rect;
}

//Adds other's size to object's size, and returns this object. (CustomRectangle&)
CustomRectangle& CustomRectangle::operator+=(const CustomRectangle& other)
{
    m_width += other.m_width;
    m_height += other.m_height;

    return (*this);
}

//Subtracts other's size from object's size and returns a CustomRectangle object called rect.
CustomRectangle CustomRectangle::operator-(const CustomRectangle &other)
{
    CustomRectangle rect;
    rect.m_width = m_width - other.m_width;
    rect.m_height = m_height - other.m_height;

    return rect;
}

//Subtracts other's size from object's size, and returns this object. (CustomRectangle&)
CustomRectangle& CustomRectangle::operator-=(const CustomRectangle &other)
{
    m_width -= other.m_width;
    m_height -= other.m_height;

    return (*this);
}

//Multiplies other's size with object's size and returns a CustomRectangle object.
```

```
CustomRectangle CustomRectangle::operator*(const CustomRectangle &other)
{
    CustomRectangle rect;
    rect.m_width = m_width * other.m_width;
    rect.m_height = m_height * other.m_height;

    return rect;
}

//Multiplies other's size with constantNumber and returns a CustomRectangle object.
CustomRectangle CustomRectangle::operator*(int constantNumber)
{
    CustomRectangle rect;
    rect.m_width = m_width * constantNumber;
    rect.m_height = m_height * constantNumber;

    return rect;
}

//Multiplies other's size with object's size, and returns this object. (CustomRectangle&)
CustomRectangle& CustomRectangle::operator*=(const CustomRectangle &other)
{
    m_width *= other.m_width;
    m_height *= other.m_height;

    return (*this);
}

//Increment operator will increase the width and height by one. Prefix Form.
CustomRectangle& CustomRectangle::operator++()
{
    ++m_width;
    ++m_height;

    return (*this);
}

//Decrement operator will decrease the width and height by one.
CustomRectangle& CustomRectangle::operator--()
{
    --m_width;
    --m_height;

    return (*this);
}

//Prints rectangle's size. This function is class' friend, so function reach all members of this class.
```

```cpp
ostream& operator<<(ostream& out, const CustomRectangle& rectangle)
{
    out<<"["<<rectangle.m_width<<"\t"<<rectangle.m_height<<"]";

    //Returns ostream&
    return out;                 //Enables cascading.
}

//Takes inputs for rectangle. This function is class' friend, so function reach all members of this class.
istream& operator>>(istream& in, CustomRectangle& rectangle)
{
    in>>rectangle.m_width;
    in>>rectangle.m_height;

    //Returns istream&
    return in;                  //Enables cascading.
}

/*****************************************
 * CustomRectangleTestMain.cpp           *
 *****************************************
 * IDE : Xcode                           *
 * Author : Şafak AKINCI                 *
 * Experiment 6: Operator Overloading    *
 *****************************************/

#include <iostream>                 //To use standart input and output functions like cin and cout.
#include "CustomRectangle.h"        //To know function prototypes of CustomRectangle.
using namespace std;                //Use cin and cout functions under the std namespace.

void TEST_Input (CustomRectangle& rect1)
{
    cout<< "+------------+" << endl
    << "| INPUT TEST |" << endl
    << "+------------+" << endl;

    //Takes inputs and they will be assigned to rect1's m_width and m_height.
    cin>>rect1;
}

void TEST_Output (CustomRectangle& rect1)
{
    cout<< "+-------------+" << endl
    << "| OUTPUT TEST |" << endl
    << "+-------------+" << endl;

    //Prints rect1's m_width and m_height.
```

```
    cout<<rect1<<endl;
}

void TEST_Addition(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+---------------+" << endl
    << "| ADDITION TEST |" << endl
    << "+---------------+" << endl;

    //Addition operator adds given rectangles, and returns final rectangle.
    //Returned rectangle is assigned to result the type of CustomRectangle.
    CustomRectangle result = rect1 + rect2;
    cout << rect1 << " + " << rect2 << " = " << result << endl;
}

void TEST_AdditionOver(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+--------------------+" << endl
    << "| ADDITION OVER TEST |" << endl
    << "+--------------------+" << endl;

    //Addition Over operator adds given rectangle to rectangle.
    cout << "Rectangle 1 Before Addition Over:\t" << rect1 << endl;
    rect1 += rect2;
    cout << "Rectangle 1 After Addition Over:\t" << rect1 << endl;
}

void TEST_Substraction(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+--------------------+" << endl
    << "|  SUBTRACTION TEST  |" << endl
    << "+--------------------+" << endl;

    //Subtraction operator subtracts given rectangle from rectangle.
    //Operator returns a rectangle the type of CustomRectangle, and it is assigned to result vector.
    CustomRectangle result = rect1 - rect2;
    cout << rect1 << " - " << rect2 << " = " << result << endl;
}

void TEST_SubstractionOver(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+-----------------------+" << endl
    << "| SUBTRACTION OVER TEST |" << endl
    << "+-----------------------+" << endl;

    //Subtraction Over operator subtracts given rectangle from rectangle.
    cout << "Rectangle 1 Before Subtraction over: " << rect1 << endl;
```

```
    rect1 -= rect2;
    cout << "Rectangle 1 After Subtraction over: " << rect1 << endl;
}

void TEST_Constant_Multiplication(CustomRectangle& rect1, double constant_value)
{
    cout<< "+-------------------------+" << endl
    << "| CONSTANT MULTIPLICATION |" << endl
    << "+-------------------------+" << endl;

    cout << "Rectangle Before Constant Multiplication:\t " << rect1 << endl;

    //Operator returns a rectangle the type of CustomRectangle, and it is assigned to result Rectangle.
    CustomRectangle result = rect1 * constant_value;
    cout << rect1 << " * " << constant_value << " = " << result << endl;
}

void TEST_Rectangle_Multiplication(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+--------------------------+" << endl
    << "| RECTANGLE MULTIPLICATION  |" << endl
    << "+--------------------------+" << endl;

    cout << "Rectangle Before Multiplication\t" << rect1 << endl;

    //Operator returns a rectangle the type of CustomRectangle, and it is assigned to result Rectangle.
    CustomRectangle result = rect1 * rect2;
    cout << rect1 << " * " << rect2 << " = " << result << endl;
}

void TEST_MultiplicationOver(CustomRectangle& rect1, CustomRectangle& rect2)
{
    cout<< "+-------------------------------+" << endl
    << "| RECTANGLE MULTIPLICATION OVER |" << endl
    << "+-------------------------------+" << endl;

    cout << "Rectangle Before Rectangle Multiplication Over: " << rect1 << endl;

    cout << rect1 << " *= " << rect2 << " = ";
    //Operator returns the object which called this operator. (CustomRectangle&)
    rect1 *= rect2;
    cout << rect1 << endl;
}


void TEST_Increment(CustomRectangle& rect1)
{
```

```
    cout<<"+-----------+"<<endl
    <<"| Increment |"<<endl
    <<"+-----------+"<<endl;

    cout<<"Rectangle before increment\t"<<rect1<<endl;

    //Rectangle's m_width and m_height will increase by one. (PREFIX FORM)
    ++rect1;
    cout<<"Rectangle after increment!\t"<<rect1<<endl;
}


void TEST_Decrement(CustomRectangle& rect1)
{
    cout<<"+-----------+"<<endl
    <<"| Decrement |"<<endl
    <<"+-----------+"<<endl;

    cout<<"Rectangle before decrement\t"<<rect1<<endl;
    //Rectangle's m_width and m_height will decrease by one. (PREFIX FORM)
    --rect1;
    cout<<"Rectangle after decrement!\t"<<rect1<<endl;
}
```

```
int main ()
{
    CustomRectangle rect1, rect2(3,4);

    TEST_Input(rect1);
    TEST_Output(rect1);
    TEST_Addition(rect1, rect2);
    TEST_AdditionOver(rect1, rect2);
    TEST_Substraction(rect1, rect2);
    TEST_SubstractionOver(rect1, rect2);
    TEST_Constant_Multiplication(rect1, 3.2);
    TEST_Rectangle_Multiplication(rect1, rect2);
    TEST_MultiplicationOver(rect1, rect2);
    TEST_Increment(rect1);
    TEST_Decrement(rect1);

    return 0;
}
```

Conclusion

→ Class' members like width and height should declare like m_width and m_height, in this case, m stands for mine. Declaration like this allow us to find this variable when we debugging.

→ Overloading operator – – and operator ++ are prefix form (--a or ++a), to declare them as postfix form, operators should take a parameter which can be integer or something. Taking a parameter will make a difference between operators.

→ Overloading operator functions can be implemented as reverse of other operator.

For example, operator >= is reverse of operator <. So in operator >= function, we can use the NOT of operator< function.

Definition of the operator like this not only enable to write less code but also make less mistake too.