# Lab 7

**Operator Overloading I**

##### Objectives

Following programming skills will be acquired in this lab:

* To understand the concept of operator overloading.
* To understand the syntax of defining an operator overloading function.
* To overload Binary and relational operators.

##### Operator Overloading

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

int a, b, c; a = b + c;

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

Class products {

string productName;

float price;

}

Products a, b, c;

a = b + c;

In fact, this will cause a compilation error, since we have not defined the behavior, our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

|  |
| --- |
| **Overloadable operators** |
| + - \* / = < > += -= \*= /= << >>  <<= >>= == != <= >= ++ -- % & ^ ! |  ~ &= ^= |= && || %= [] () , ->\* -> new delete new[] delete[] |

To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

type operator sign (parameters) { /\*...\*/ }

Here you have an example that overloads the addition operator (+). We are going to create a class to store bidimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bidimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).

##### Example 7.1

|  |  |
| --- | --- |
| #include <iostream> using namespace std;  class **Vector** { public:  int x,y; Vector () {};  Vector (int,int);  Vector operator + (Vector);  };  **Vector::Vector (int a, int b**)  { x = a;  y = b;  }  **Vector Vector::operator+ (Vector param)** { Vector temp;  temp.x = x + param.x; temp.y = y + param.y; return (temp);  }  int **main** () { Vector a (3,1);  Vector b (1,2);  Vector c;  **c = a + b;**  cout << c.x << "," << c.y;  return 0;  } | **Output:**  4,3 |

It may be a little confusing to see so many times the Vector identifier. But, consider that some of them refer to the class name (type) Vector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

Vector (int, int); // function name Vector (constructor) Vector operator+ (CVector); // function returns a Vector

The function operator+ of class Vector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

c = a + b;

c = a.operator+ (b);

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

Vector () { };

This is necessary, since we have explicitly declared another constructor:

Vector (int, int);

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves to be able to construct objects of this type without parameters. Otherwise, the declaration:

Vector c;

included in main() would not have been valid.

An empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

Vector () { x=0; y=0; };

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

Vector d (2,3); Vector e;

e = d; // copy assignment operator

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if

you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Operator** | **Member function** | **Global function** |
| @a | + - \* & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - \* / % ^ & | < > == != <= >= << >> && || , | A::operator@ (B) | operator@(A,B) |
| a@b | = += -= \*= /= %= ^= &= |= <<= >>= [] | A::operator@ (B) | - |
| a(b, c...) | () | A::operator() (B, C...) | - |
| a->x | -> | A::operator->() | - |

Where a is an object of class A, b is an object of class B and c is an object of class C.

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

##### Example 7.2

for c1 + c2

shorthand notation

//

// instantiate complex c1

// instantiate complex c2

int **main**()

{

complex c1(2.2,3.0);

complex c2(1.0,-4.5); complex c3=c1+c2; c3.show();

return 0;

}

// Complex number class with operator overloading

#include <iostream> using namespace std;

class **complex**

{

float re, im; // real and imaginary parts public:

complex () { re = im = 1;}

complex (float r, float i) { re = r; im = i;};

**complex operator+(complex c);**

void show()

{ cout << "(" << re << "," << im << ")" << endl; }

};

complex complex::**operator+**(complex c)

{ complex sum;

sum.re = re + c.re; sum.im = im + c.im; return sum;

}

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

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

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

**Example 7.3**

[Live Demo](http://tpcg.io/JbI9Wt)

#include <iostream>

using namespace std;

class Distance {

private:

int feet; // 0 to infinite

int inches; // 0 to 12

public:

// required constructors

Distance() {

feet = 0;

inches = 0;

}

Distance(int f, int i) {

feet = f;

inches = i;

}

// method to display distance

void displayDistance() {

cout << "F: " << feet << " I:" << inches <<endl;

}

// overloaded minus (-) operator

Distance operator- () {

feet = -feet;

inches = -inches;

return Distance(feet, inches);

}

// overloaded < operator

bool operator <(Distance& d) {

if(feet < d.feet) {

return true;

}

if(feet == d.feet && inches < d.inches) {

return true;

}

return false;

}

};

int main() {

Distance D1(11, 10), D2(5, 11);

if( D1 < D2 ) {

cout << "D1 is less than D2 " << endl;

} else {

cout << "D2 is less than D1 " << endl;

}

return 0;

}

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

D2 is less than D1

**Practical Exercises**

**Exercise 7.1**

Extend the Vector class by providing operator overloading functions to perform the following operations:

* - for subtraction of two Vector objects
* \* for multiplication of a Vector by a scalar (i.e with some int value)

Write a driver program (main() ) to test your class. Assume data members (int x, int y). write getter, setter functions.

Code:

#include <iostream>

using namespace std;

class Vector {

int x, y;

public:

Vector(int, int);

Vector operator + (Vector);

Vector operator - (Vector);

Vector operator \* (int a);

void setX(int a) {

x = a;

}

void setY(int a) {

y = a;

}

int getX() {

return x;

}

int getY() {

return y;

}

};

Vector::Vector(int a=0, int b=0)

{

setX(a);

setY(b);

}

Vector Vector::operator+ (Vector param) {

Vector temp;

temp.x = x + param.x;

temp.y = y + param.y;

return (temp);

}

Vector Vector::operator- (Vector param) {

Vector temp;

temp.x = x - param.x;

temp.y = y - param.y;

return (temp);

}

Vector Vector::operator\* (int a) {

Vector temp;

temp.x = x \* a ;

temp.y = y \* a;

return (temp);

}

int main() {

Vector a(3, 1);

Vector b(1, 2);

Vector c;

c = a + b;

cout << c.getX() << "," << c.getY() << endl;

c = a - b;

cout << c.getX() << "," << c.getY()<< endl;

c = a \* 5;

cout << c.getX() << "," << c.getY()<< endl;

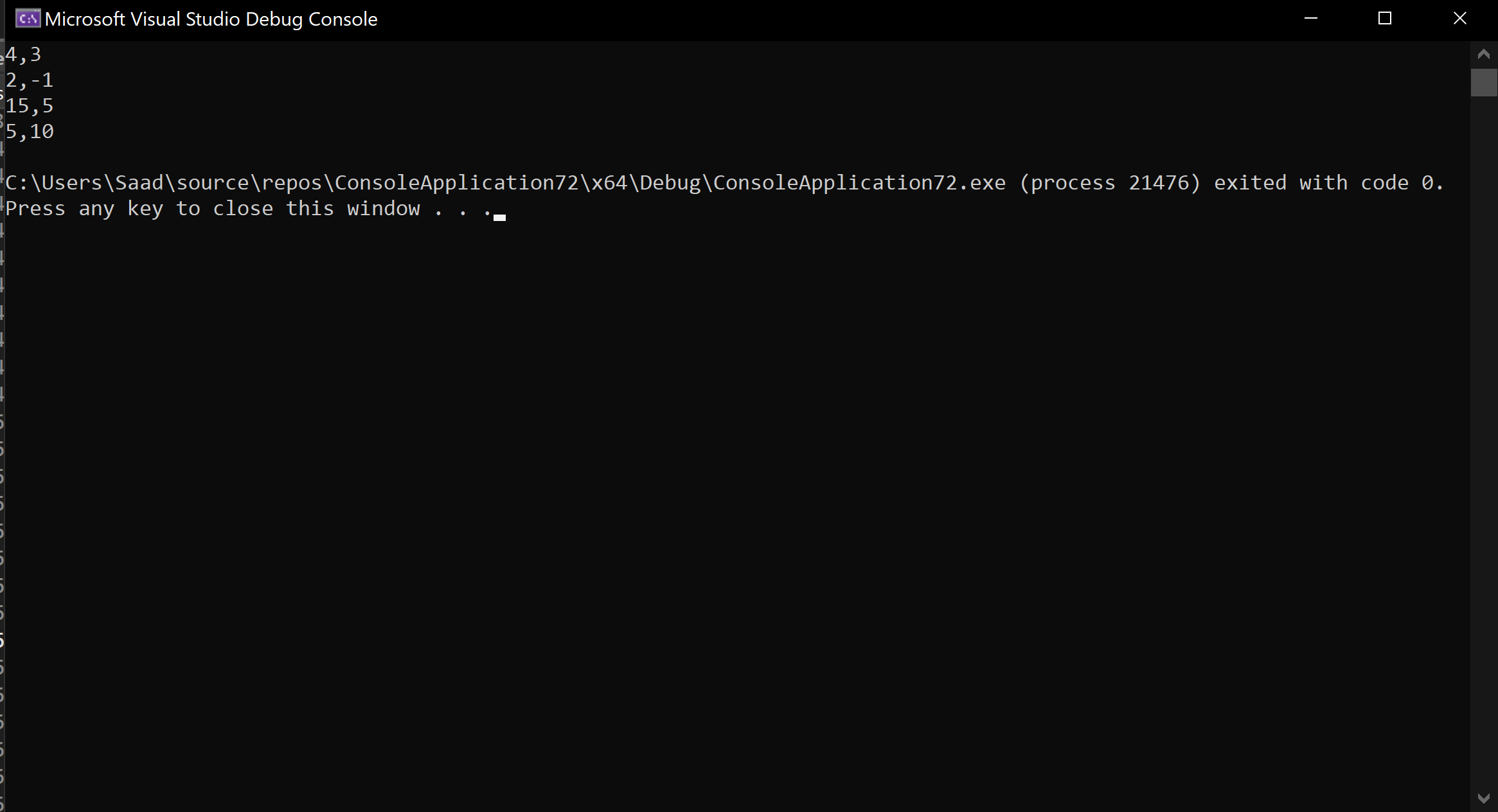
c = b \* 5;

cout << c.getX() << "," << c.getY() << endl;

return 0;

}

Output:



**Exercise 7.2**

Extend the Complex class (Example 6.2) by providing operator overloading functions to perform the following operations:

* - for subtraction of two Complex numbers
* \* for multiplication of two Complex numbers Write a driver program to test your class. Assume (float real, float imag) data members.

Code:

#include <iostream>

using namespace std;

class Complex {

float real;

float img;

public:

Complex(float a = 0, float b = 0) {

real = a;

img = b;

}

Complex operator+(Complex a1) {

Complex a3;

a3.real = a1.real + real;

a3.img = a1.img + img;

return a3;

}

Complex operator-(Complex a2) {

Complex a3;

a3.real = real - a2.real;

a3.img = img - a2.img;

return a3;

}

Complex operator\*(Complex a1) {

int a, b, c;

Complex a3;

a = a1.real \* real;

b = a1.img \* img;

c = a - b;

a3.real = c;

a = a1.real \* img;

b = a1.img \* real;

c = a + b;

a3.img = c;

return a3;

}

void getInput() {

cout << "Enter the real part : " << endl;

cin >> real;

cout << "Enter the imaginary part : " << endl;

cin >> img;

}

void displyAdd() {

cout << "Sum of two complex numbers is " << endl;

cout << real << "+" << img << "i" << endl;

}

void displySub() {

cout << "Difference of two complex numbers is " << endl;

cout << real << "+" << img << "i" << endl;

}

void displyMul() {

cout << "Product of two complex numbers is " << endl;

cout << real << "+" << img << "i" << endl;

}

};

int main() {

Complex a1, a2, a3;

char choice;

a1.getInput();

a2.getInput();

do {

cout << "Enter + for addition, - for subtraction, \* for multiplication" << endl;

cin >> choice;

switch (choice) {

case '+': a3 = a1+a2;

a3.displyAdd();

break;

case '-': a3 = a1-a2;

a3.displySub();

break;

case '\*': a3 = a1\*a2;

a3.displyMul();

break;

default:

cout << "Invalid option" << endl;

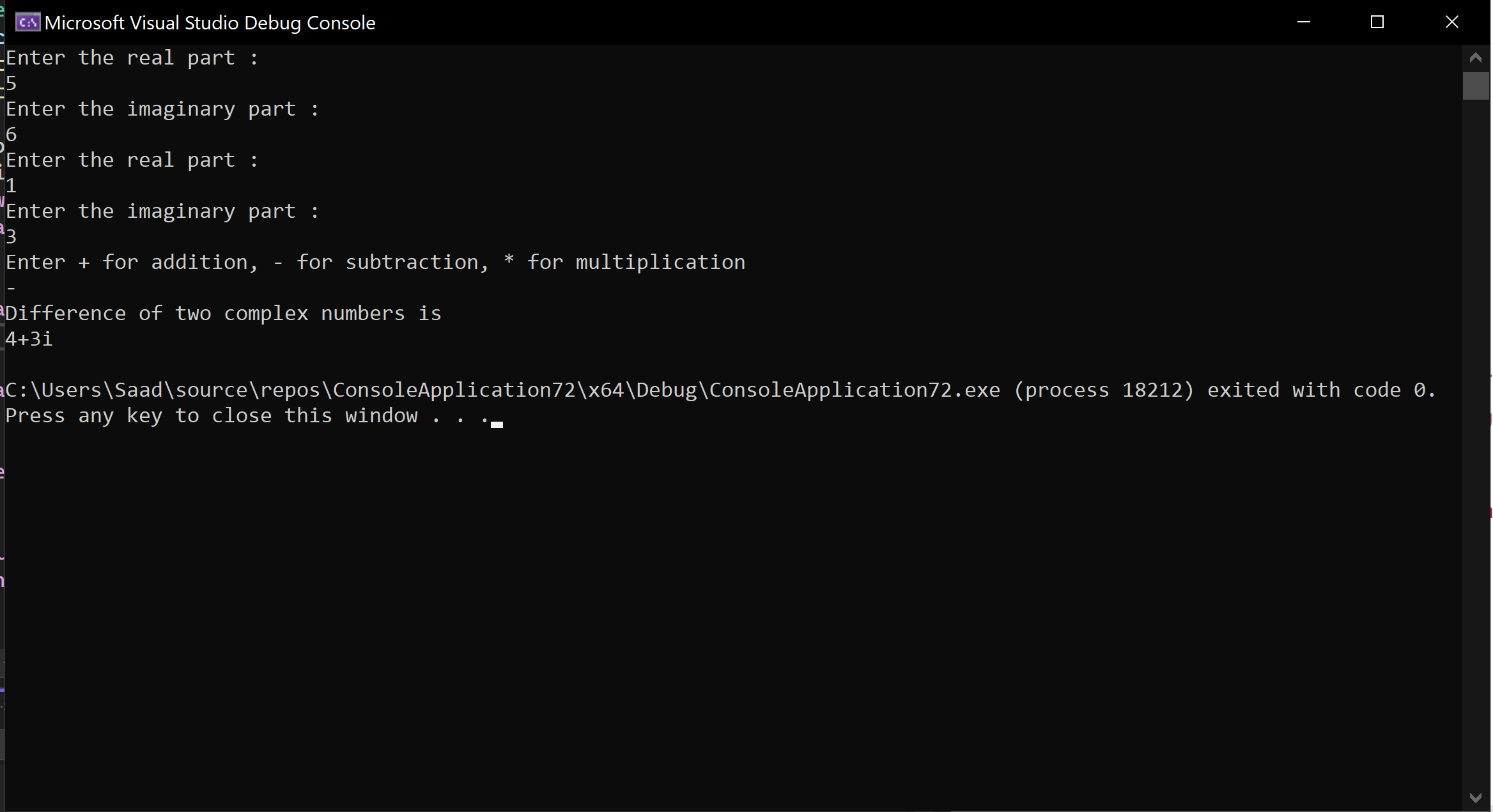
}

} while (choice != '+' && choice != '-' && choice != '\*');

return 0;

}

Output:



**Exercise 7.3**

Modify the Rectangle class (as you have seen in Lab Journal 3: Example 3.3) to overload the + operator so that you can add two Rectangle objects. Adding two Rectangle objects should give another Rectangle object whose length is the sum of the lengths of the two Rectangle objects and whose breadth is the sum of the breadths of the two Rectangle objects. Assume (int length, int breadth) data members.

Code:

#include <iostream>

using namespace std;

class Rectangle

{

private:

int length, breadth;

public:

void setValues(int a, int b)

{

length = a;

breadth = b;

}

Rectangle operator+(Rectangle a) {

Rectangle c;

c.length = length + a.length;

c.breadth = breadth + a.breadth;

return c;

}

void display() const {

cout <<"Length : " << length << endl;

cout << "Breadth : " << breadth << endl;

}

};

int main()

{

Rectangle rect1,rect2,rect3;

rect1.setValues(3, 4);

rect2.setValues(10, 12);

rect3 = rect1 + rect2;

rect3.display();

return 0;

}

Output :

