# Lab 8

**Operator Overloading II**

##### Objectives

Following programming skills will be acquired in this lab:

* To understand the concept of operator overloading with friend functions.
* To understand the scenarios of using friend classes.
* To understand and implement the scenario of composition with friend classes.

##### 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 8.1

|  |  |
| --- | --- |
| #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).

**Lab Journal Questions**

**Q # 1: Write code for following versions of main().**

1. void main()

{

Point p1(5, 11.5);

Point p2;

P2.input();

If(p1 < p2)

Cout<<”p1 is less than p2”;

Else

Cout<<”p1 is greater than p2”;

getch();

}

Code: #include <iostream>

using namespace std;

class Point {

int x;

int y;

public:

Point(int a=0,int b=0){

x = a;

y = b;

}

void input() {

cout << "Enter x and y " << endl;

cin >> x;

cin >> y;

}

bool operator<(Point &p1) {

if (p1.x > x && p1.y > y) {

return true;

}

else {

return false;

}

}

};

int main() {

Point p1(5, 11.5);

Point p2;

p2.input();

if(p1 < p2)

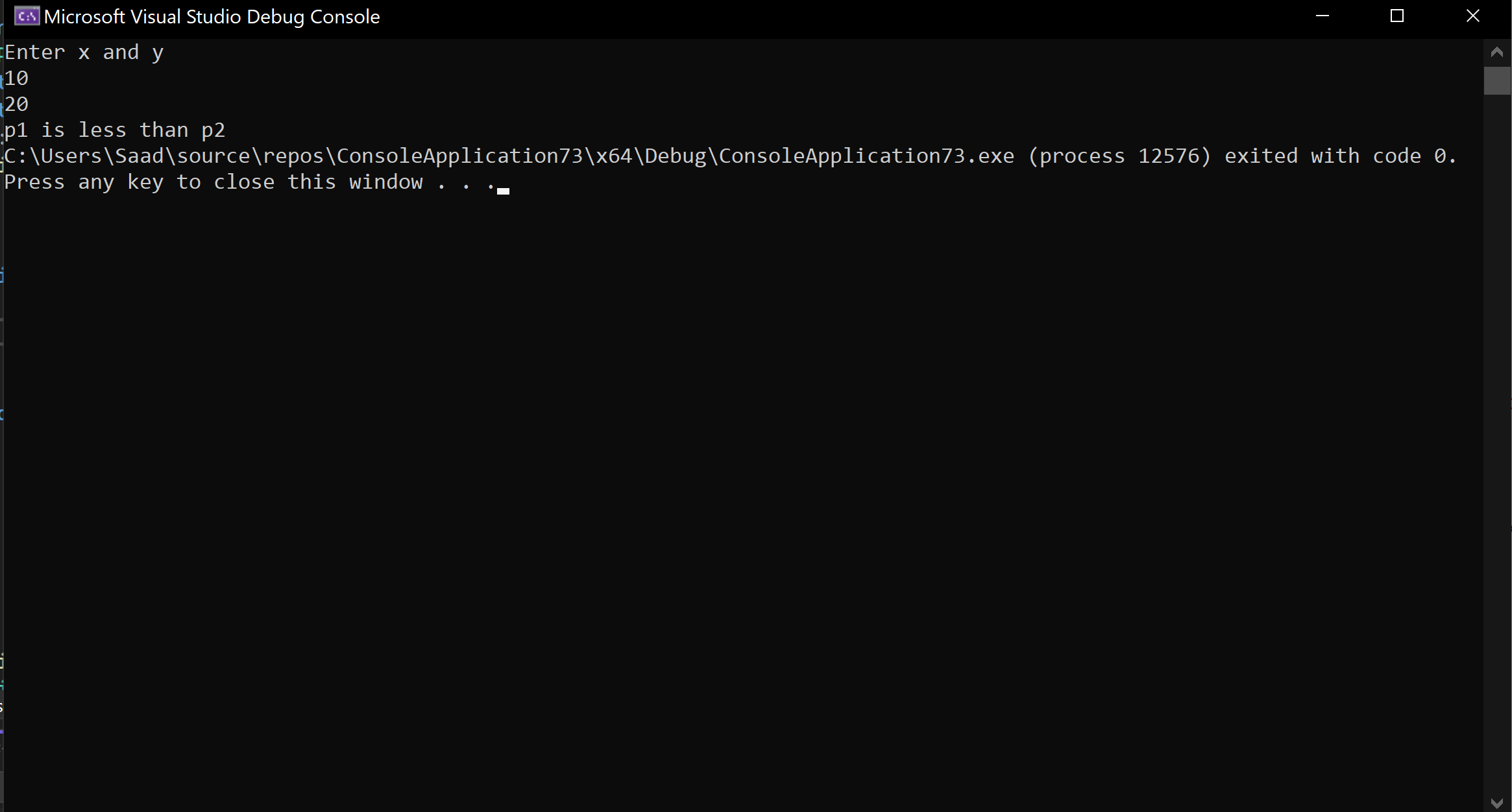
cout << "p1 is less than p2";

else

cout << "p1 is greater than p2";

}

Output:



**Modify main() as following.**

1. void main()

{

Point p1(5, 11.5);

Point p2, p3;

P2.input();

P3 = p1 < p2;

P3.show();

getch();

}

Code:

#include <iostream>

using namespace std;

class Point {

int x;

float y;

public:

Point(int a=0,float b=0){

x = a;

y = b;

}

void input() {

cout << "Enter x and y " << endl;

cin >> x;

cin >> y;

}

Point operator<(Point &p1) {

if (p1.x > x && p1.y > y) {

return p1;

}

else {

return \*this;

}

}

void show() const {

cout << "x = " << x << endl;;

cout << "y = " << y << endl;

}

};

int main() {

Point p1(5, 11.5);

Point p2, p3;

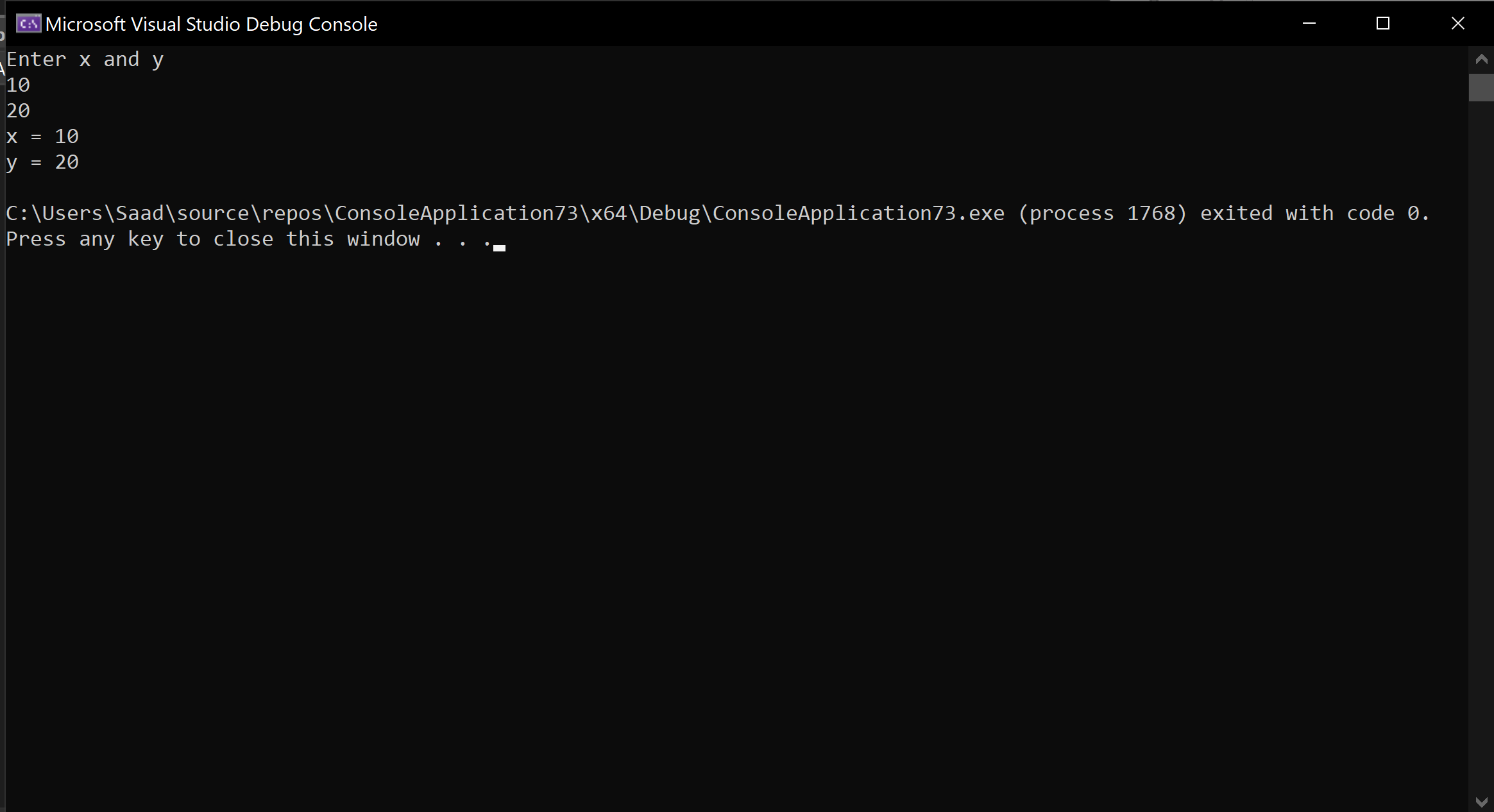
p2.input();

p3 = p1 < p2;

p3.show();

}

Output:



**Revise main() as following.**

1. void main()

{

Point p1(5, 11.5);

P1++;

}

Code:

#include <iostream>

using namespace std;

class Point {

int x;

float y;

public:

Point(int a=0,float b=0){

x = a;

y = b;

}

void operator++(int) {

x++;

y++;

}

void show()const {

cout << "x = " << x << endl;

cout << "y = " << y << endl;

}

};

int main() {

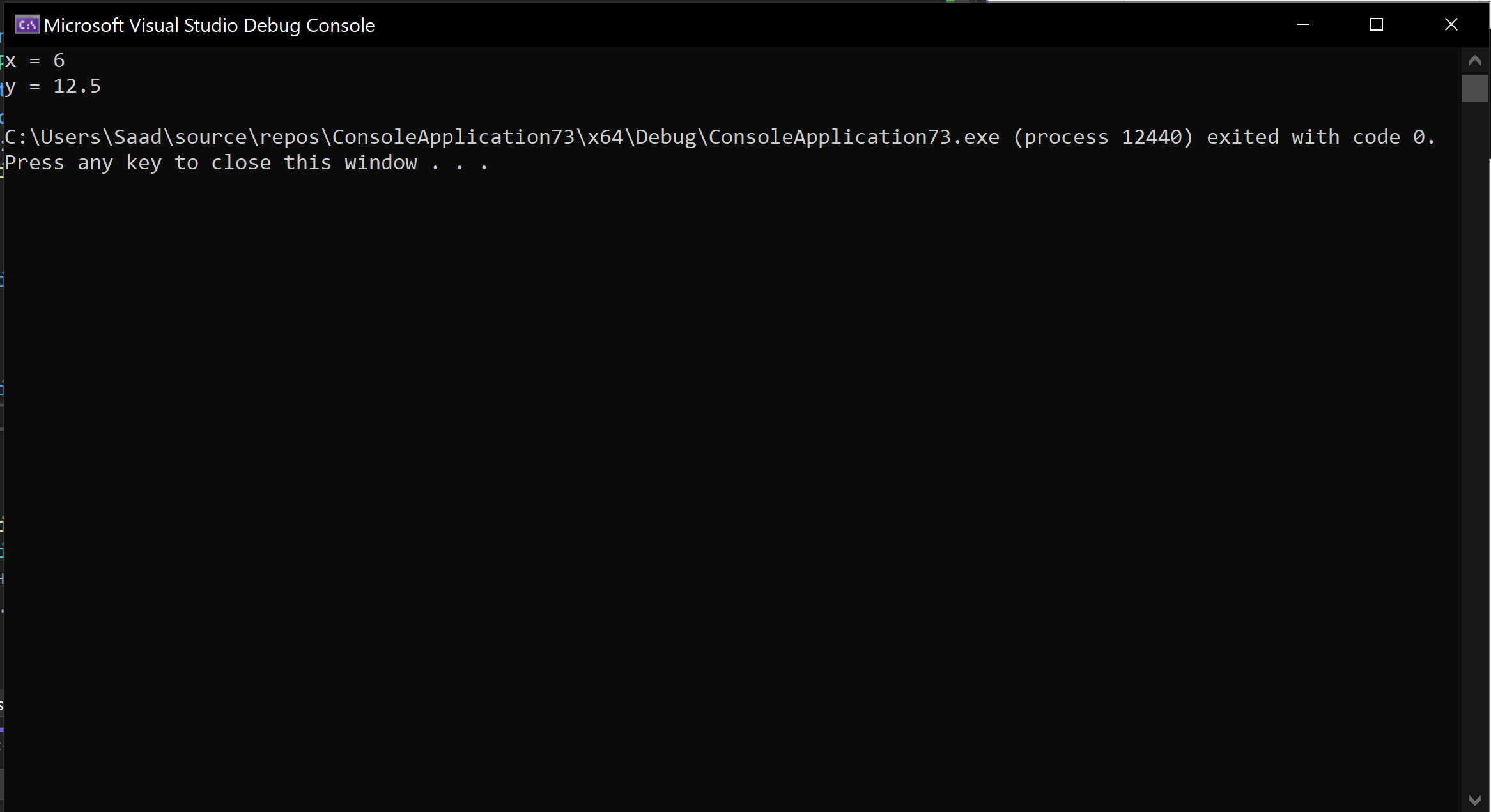
Point p1(5, 11.5);

p1++;

p1.show();

}

Output:



**Write non-member function for operator overloading in following main().**

1. void main()

{

Point p1(5, 11.5);

Point p2;

P2.input();

P1 + p2; //non-member function

getch();

}

Code:

#include <iostream>

using namespace std;

class Point {

int x;

float y;

public:

Point(int a=0,float b=0){

x = a;

y = b;

}

void operator+(Point a) {

a.x= x + a.x;

a.y = y + a.y;

cout << "x = " << a.x << " y = " << a.y << endl;

}

void input() {

cout << "Enter x and y : " << endl;

cin >> x;

cin >> y;

}

};

int main() {

Point p1(5, 11.5);

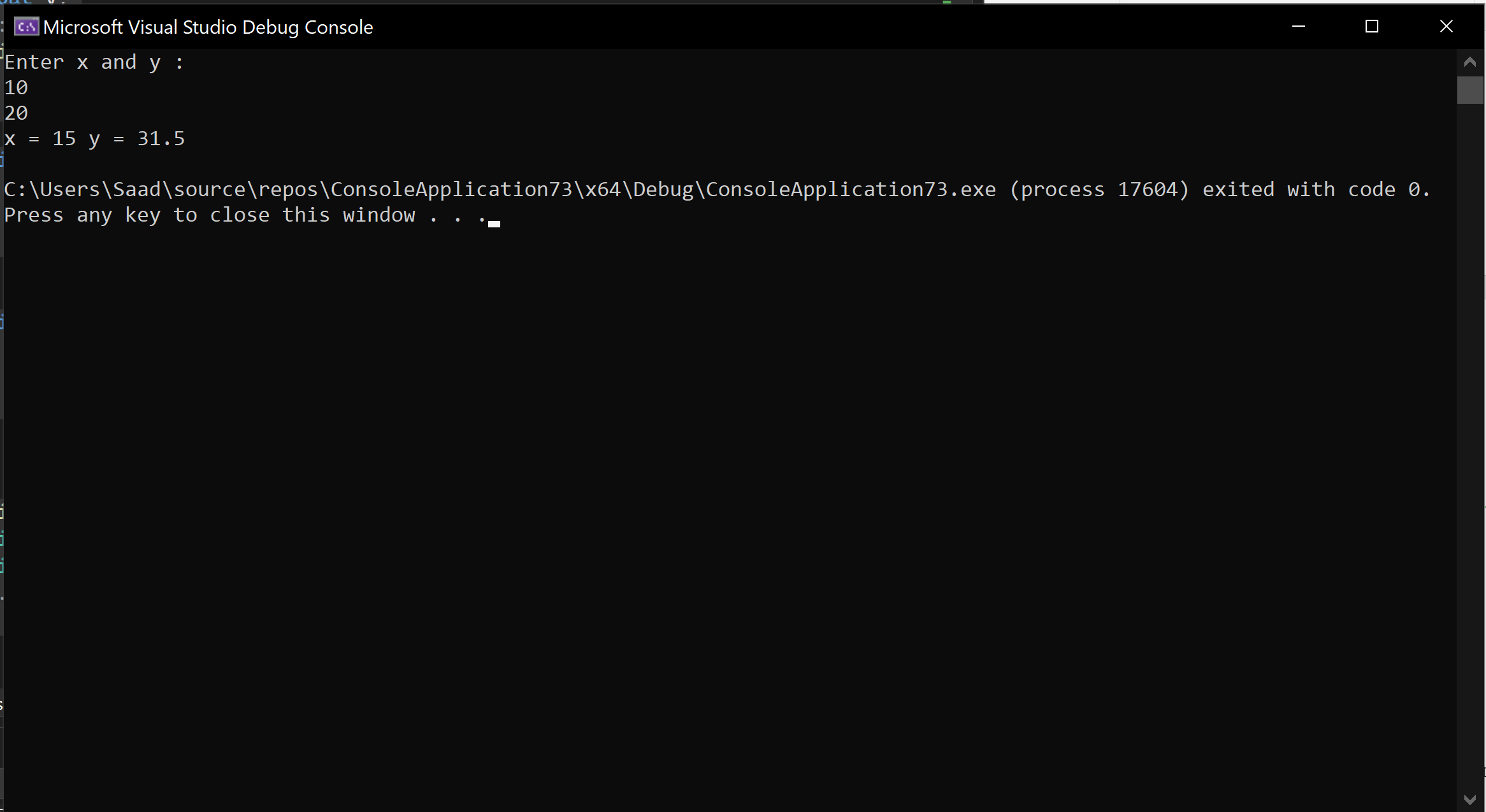
Point p2;

p2.input();

p1 + p2; //non-member function

}

Output:



**Write non-member friend function for operator overloading in following main().**

1. void main()

{

Point p1(5, 11.5);

Point p2;

P2.input();

P1 + p2; //non-member function

getch();

}

Code:

#include <iostream>

using namespace std;

class Point {

int x;

float y;

public:

friend void operator+(Point a, Point b);

Point(int a=0,float b=0){

x = a;

y = b;

}

void input() {

cout << "Enter x and y : " << endl;

cin >> x;

cin >> y;

}

};

void operator+(Point a,Point b) {

a.x = b.x + a.x;

a.y = b.y + a.y;

cout << "x = " << a.x << " y = " << a.y << endl;

}

int main() {

Point p1(5, 11.5);

Point p2;

p2.input();

p1 + p2; //non-member function

}

Ouput:

