# Part1: Operator overloading

Define a class named Box with 3 member variables of type float length, breadth and height, and

the following member functions:

Define a class named Box with 3 member variables of type float length, breadth and height, and

the following member functions:

Define a class named Box with 3 member variables of type float length, breadth and height, and

the following member functions:

The method of extending the operation on user-defined types (objects) is called operator overloading. Operators can be overloaded for different operations. Most programmers implicitly use overloaded operators regularly. For example, the addition operator (+) operates quite differently on integers, float and doubles, and other built-in types because the operator (+) has been overloaded in the C++ language itself and with some library objects such as strings. Operators are overloaded by writing a function definition as we normally do, except that the function name now becomes the keyword operatorfollowed by the symbol for the operator being overloaded. Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators for our class.

The syntax of operator overloading is as follows:

return\_type operator operator\_symbol (parameters)

{

statements;

}

Example:

#include <iostream>

using namespace std;

class A

{

private:

int x; int y; char f;

public:

A(int x1 = 0, int y1 = 0) { x = x1; y = y1; }

void setx(int x1) { x = x1; }

void sety(int y1) { y = y1; }

int getx() { return x; }

int gety() { return y; }

A operator +(A r1)

{

A nr;

nr.x = x + r1.getx();

nr.y = y + r1.gety();

return nr;

}

};

int main()

{

A a1(3, 4), a2(5, 5), a3;

a3 = a1 + a2;

cout<<a3.getx()<<endl;

cout<<a3.gety()<<endl;.

}

Output:

|  |
| --- |
|  |

Now, update the above class by adding the following operator:

bool operator >=(A r1)

{

if (x >= r1.getx() && y >= r1.gety())

return true;

else

return false;

}

Similarly, overload the operator = to assign one value of a complex object to the other complex object.

Part2: **Friend Function and Class**

In some cases, we need to access the private data members of a class from non-member functions. In such situations, we must declare the function as a friend function of the class. This friend function seems to violate the data hiding feature of the OOP concept. However, the function that accesses private data must be declared as a friend function within the class. With friend functions, data integrity is still maintained.

Sometimes we may need to make, one or all the member functions of a class friend to another class. For that, we declare a class or member function as a friend to the other class so that one or all the member functions of the declared class can access the private members of the former class.

Example:

// C++ program to demonstrate the working of friend function

#include <iostream>

using namespace std;

class Distance {

private:

int meter;

// friend function

friend int addFive(Distance);

public:

Distance() : meter(0) {}

};

// friend function definition

int addFive(Distance d) {

//accessing private members from the friend function

d.meter += 5;

return d.meter;

}

int main() {

Distance D;

cout << "Distance: " << addFive(D);

return 0;

}

Output:

|  |
| --- |
|  |

Similarly, add another friend to convert meter to centimeter.

Exercises:

1. Write a class Date that overloads prefix and postfix operators to increase the Date object by one day, while causing appropriate increments to the month and year (use the appropriate condition for leap year). The prefix and postfix operators in the Date class should behave exactly like the built-in increment operators.
2. Write a class to store x, y, and z coordinates of a point in three-dimensional space. Overload addition and subtraction operators for addition and subtraction of two coordinate objects. Implement the operator functions as non-member functions (friend operator functions).

That is the end of labsheet.. Good Luck