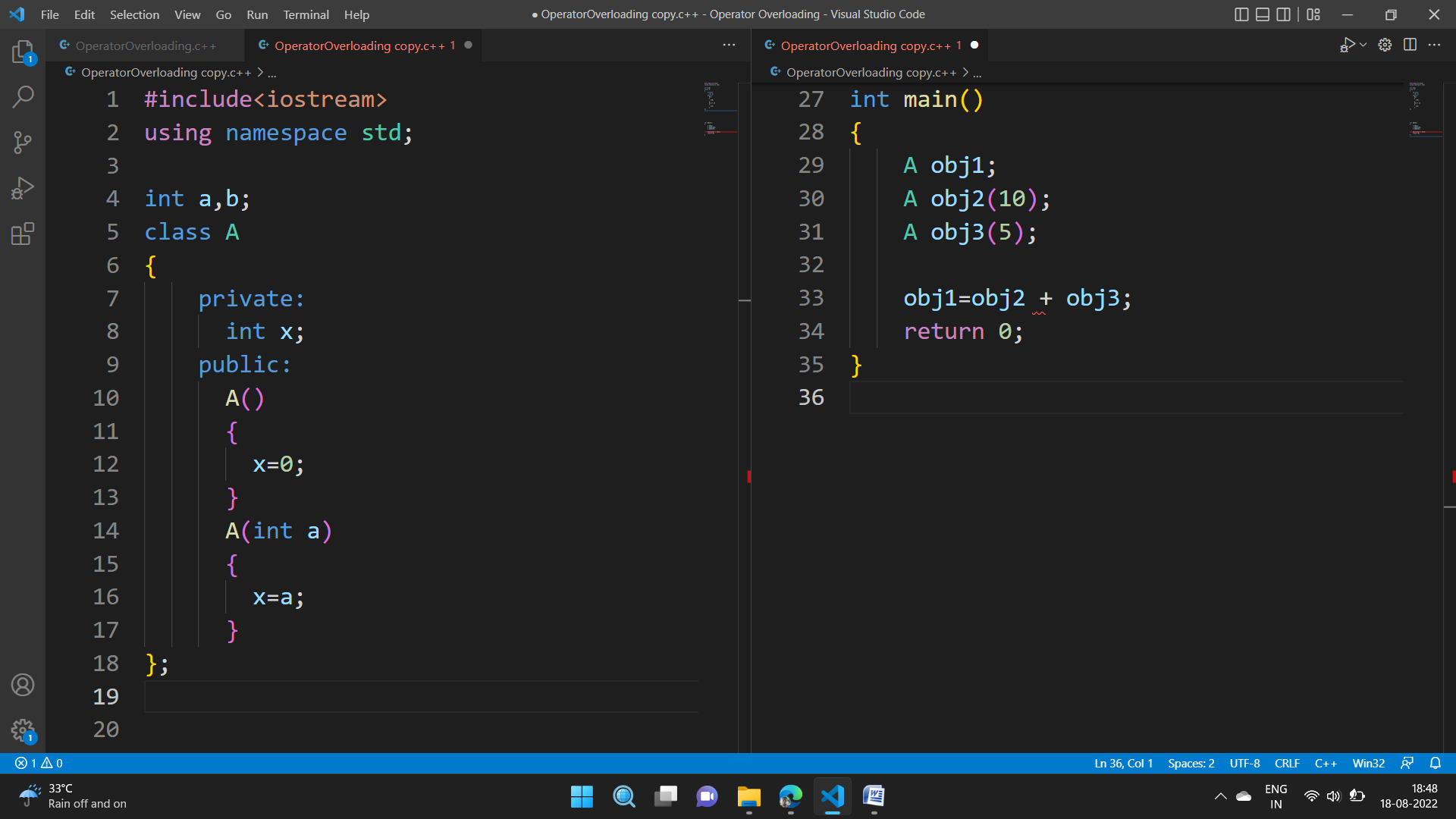
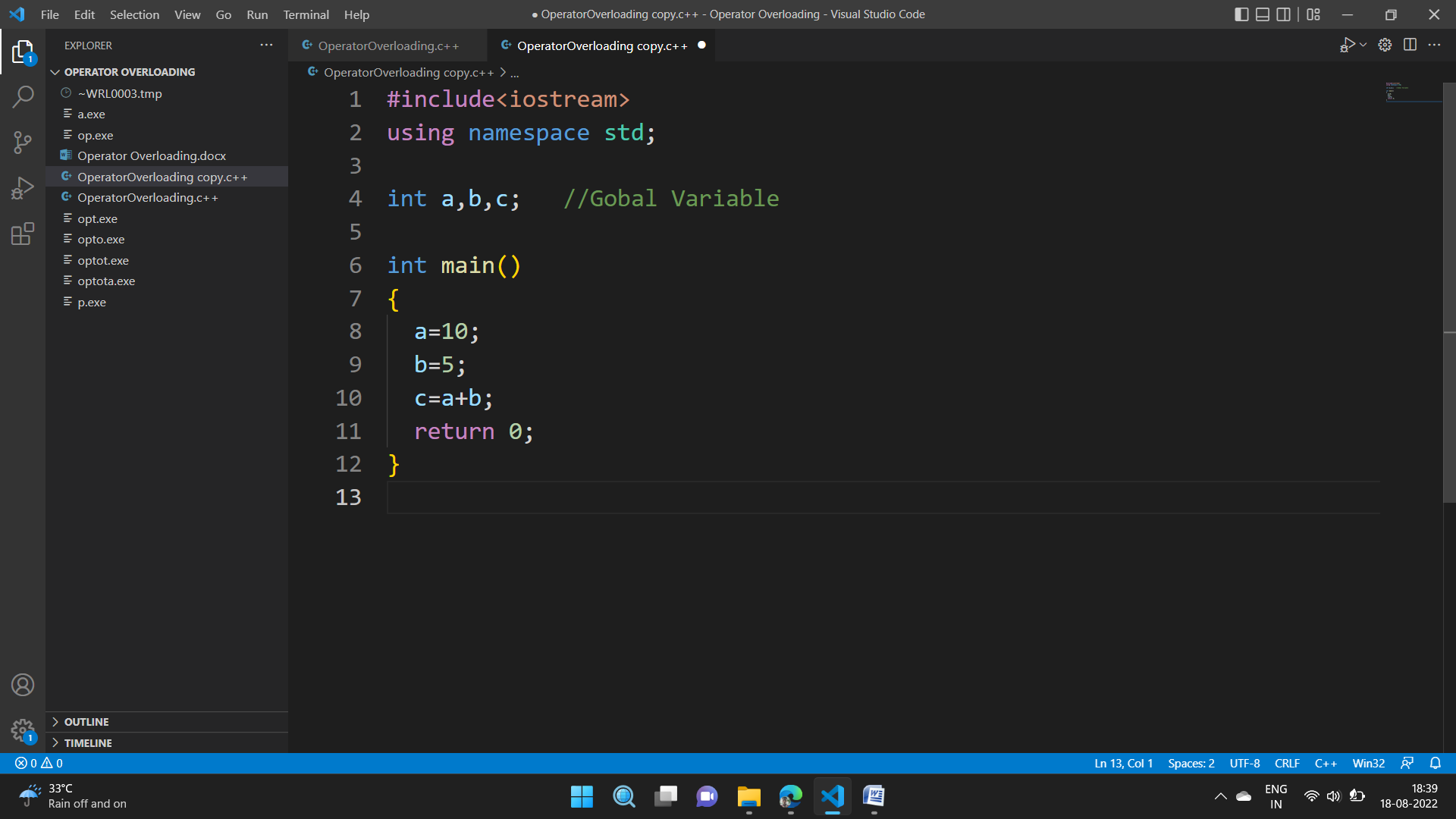
**Operator Overloading**

****

**Compile time error**

**//Compile time error**

**A obj1**

**no operator "+" matches these operandsC/C++(349)**

**operator\_overloading.c++(38, 13): operand types are: A + A**

**Here, variables “a” and “b” are of types “int”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the addition operator “+” is predefined to add variables of built-in data type only.**

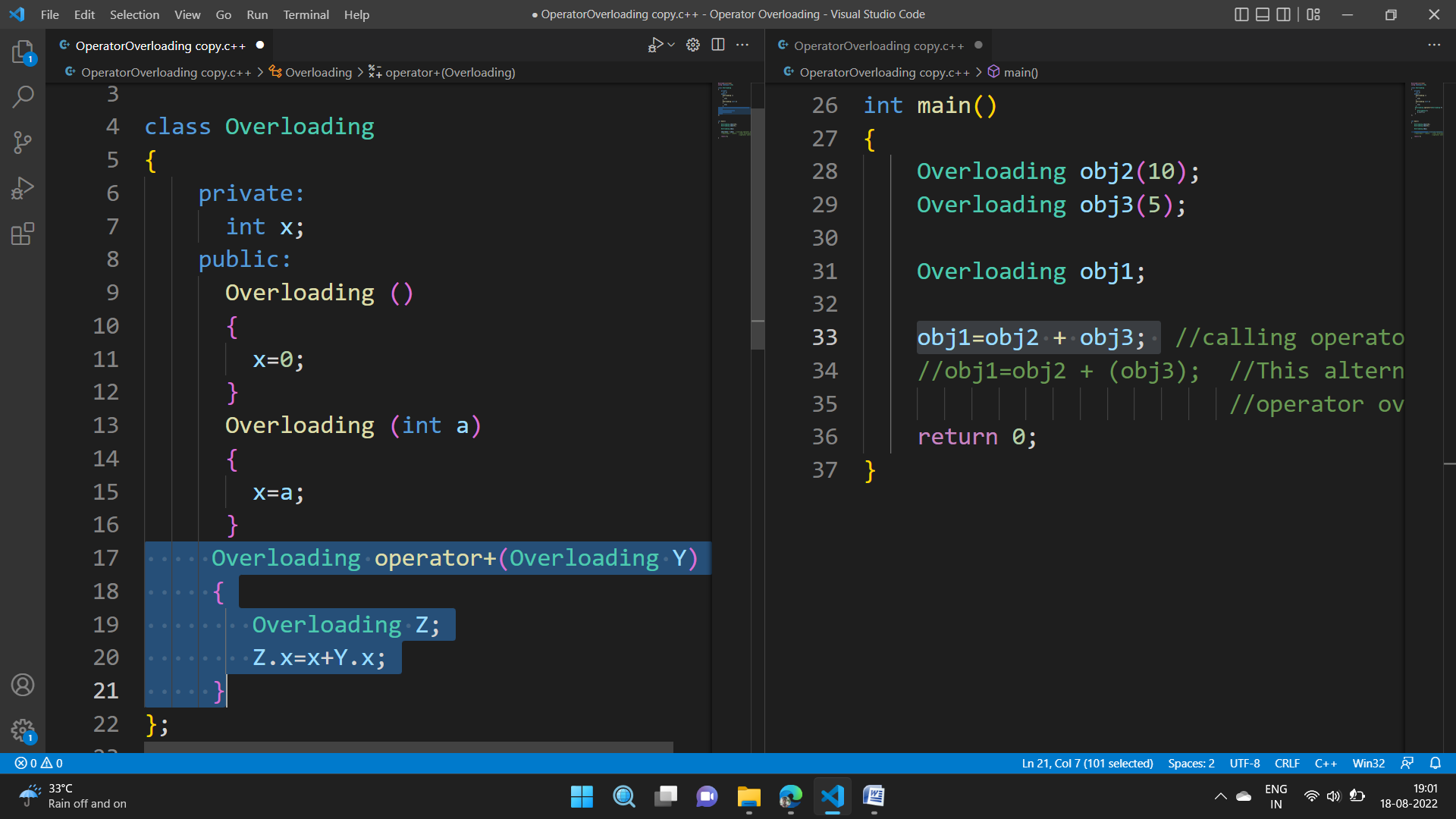
**Now, consider another example, we have 3 variables “obj1”, “obj2” and “obj3”of type “class A”.**

**Here we are trying to add two objects “obj2” and “obj3”, which are of user-defined type of type “class A” using the “+” operator, we will get an error, because the compiler don’t know how to add two objects and the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.**

**Operator Overloading:**

**Operator overloading is used to perform the operation on the use-defined data type in much the same way as the built-in type.**

* **It is a compile time polymorphism.**
* **Operator overloading is used to overload or redefines most of the operators available in C++.**
* **When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked.**
* **Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.**

****

* **The left hand side operand (obj2) is used to invoke the operator function and the right hand side operand (obj3) is passed as an argument.**
* **We know that a member function can be invoked only by an object of the same class. Here obj2 is used to invoke the operator function and the right hand side operand (obj3) is passed as an argument. The above invocation statement is equivalent to**

**obj1=obj2.operator+(obj3);**

* **obj1=obj2+obj3 the compiler treat it as obj1=obj2.add(obj3).**
* **obj1=obj2+obj3+obj4, hear we have 3 object to add.**

**First the addition of object obj3 and obj4 will happen (obj2+obj3+obj4) then the result of this two will add with obj2.**

**Operator Overloading Syntax:**

**return type class name :: operator operator symbol(parameter list)**

**{**

**}**

**Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.**

**Rules for Operator Overloading:**

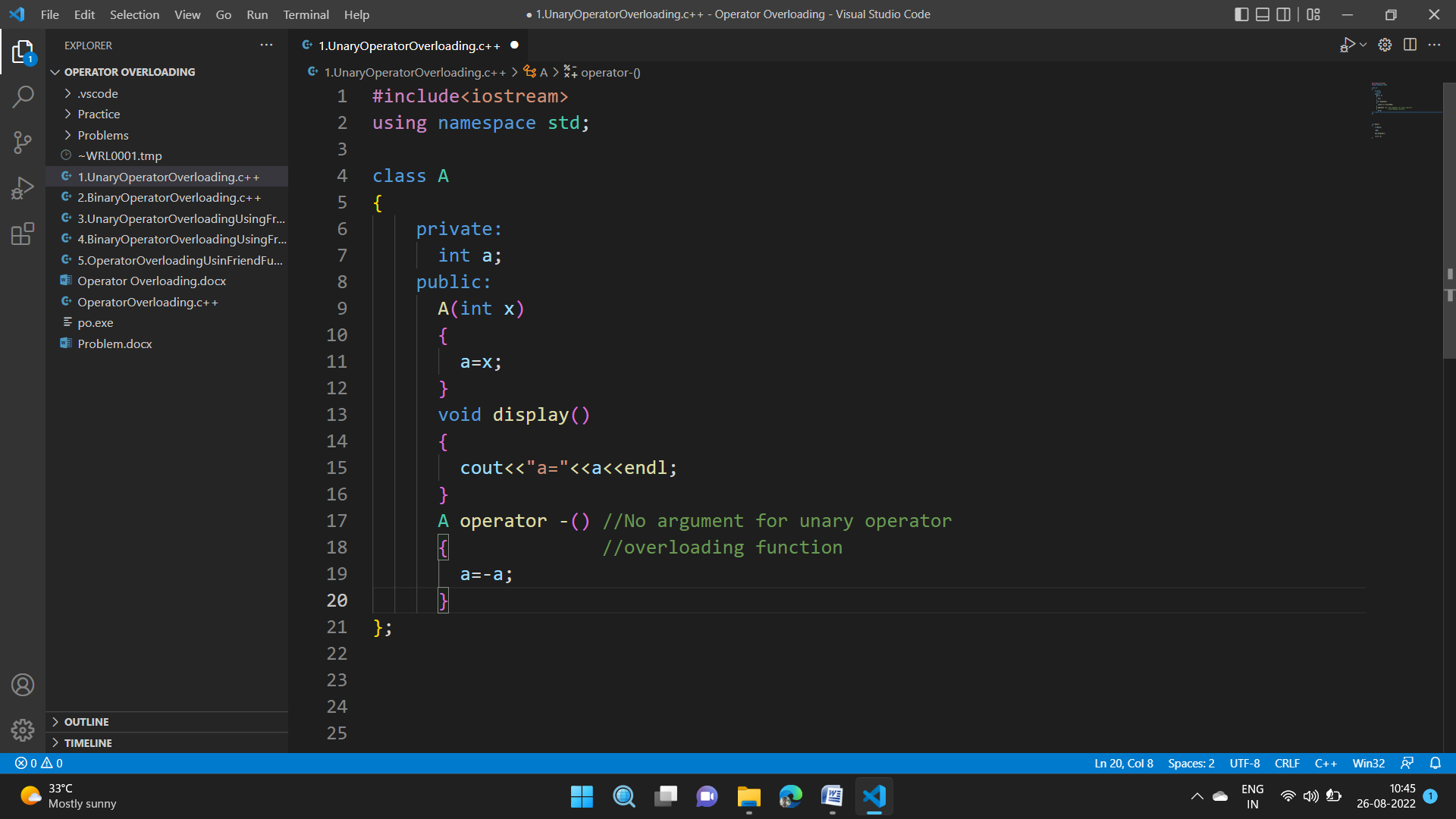
* **For operator overloading to work, at least one of the operands must be a user-defined class object.**
* **Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function(friend function).**
* **We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.**
* **When unary operators are overloaded through a member function take no explicit arguments, and if unary operators are overloaded through a friend function, takes one explicit argument.**
* **When binary operators are overloaded through a member function takes one explicit argument, and if binary operators are overloaded through a friend function takes two explicit arguments.**
* **Binary arithmetic operator such as +, -, \* and / must be explicitly return a value. They must not attempt to change their own arguments.**

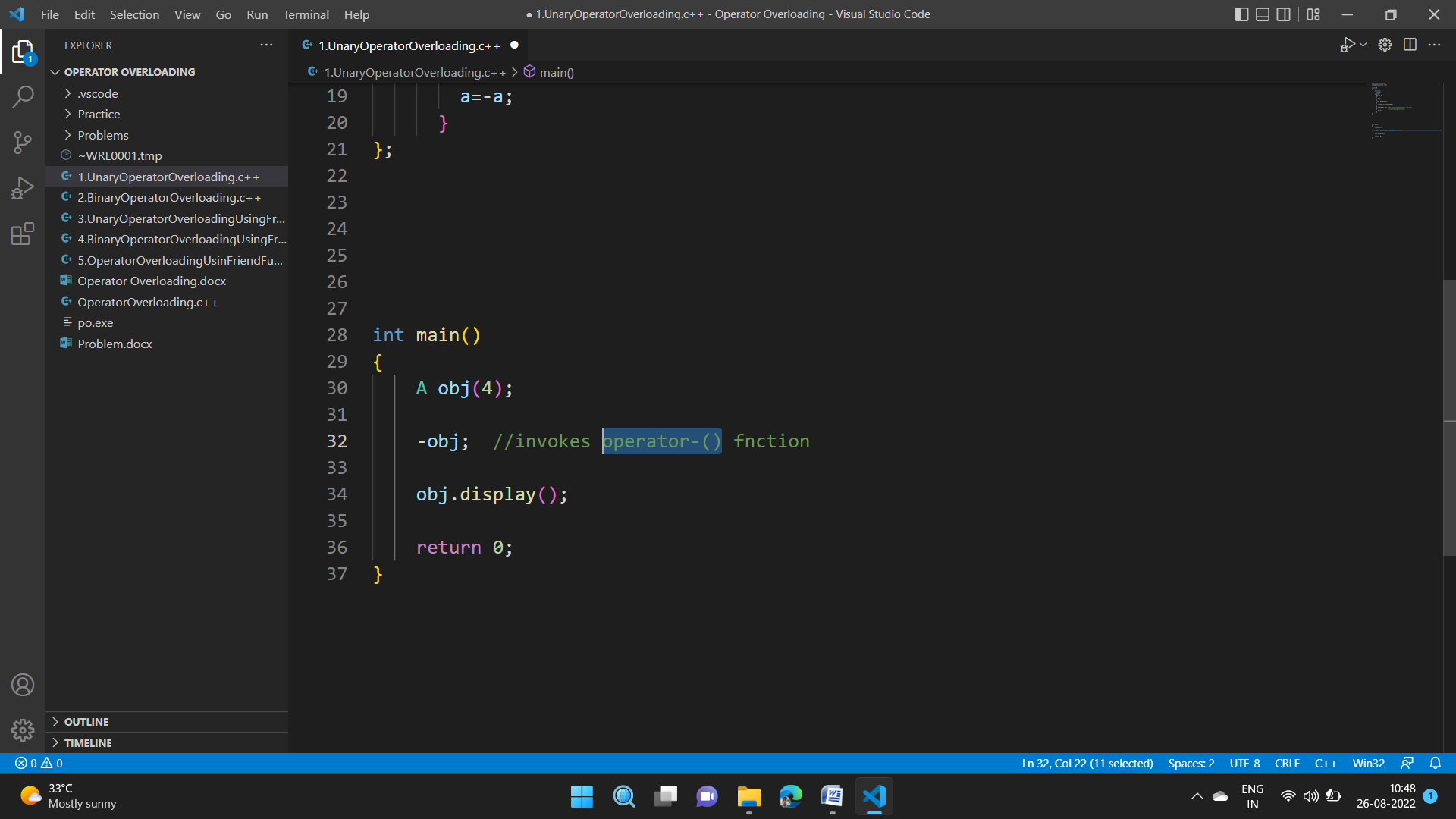
**Operator overloading can be perform in two ways:**

**Operator overloading can be perform in two ways**

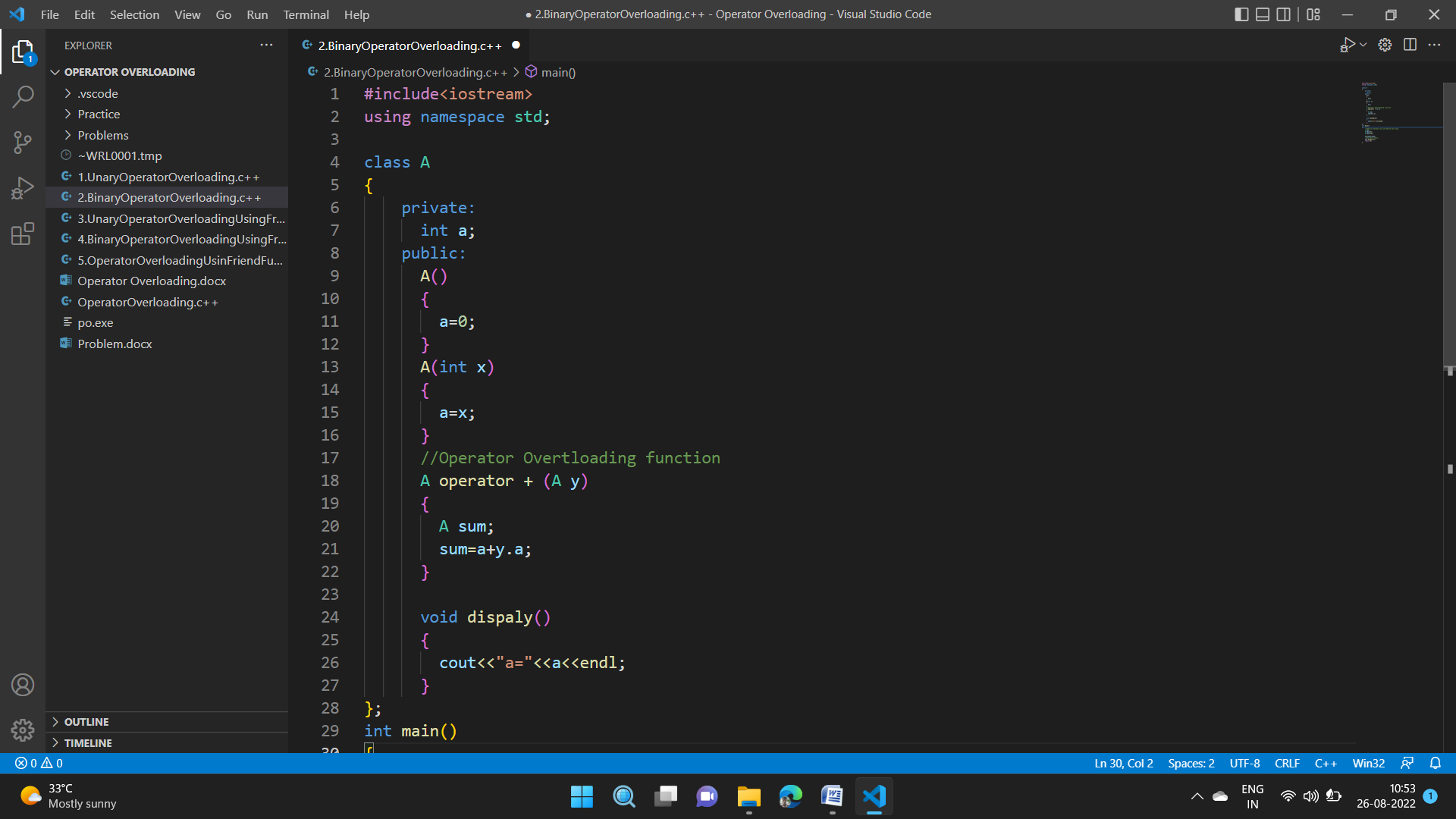
**Using Member Function:**

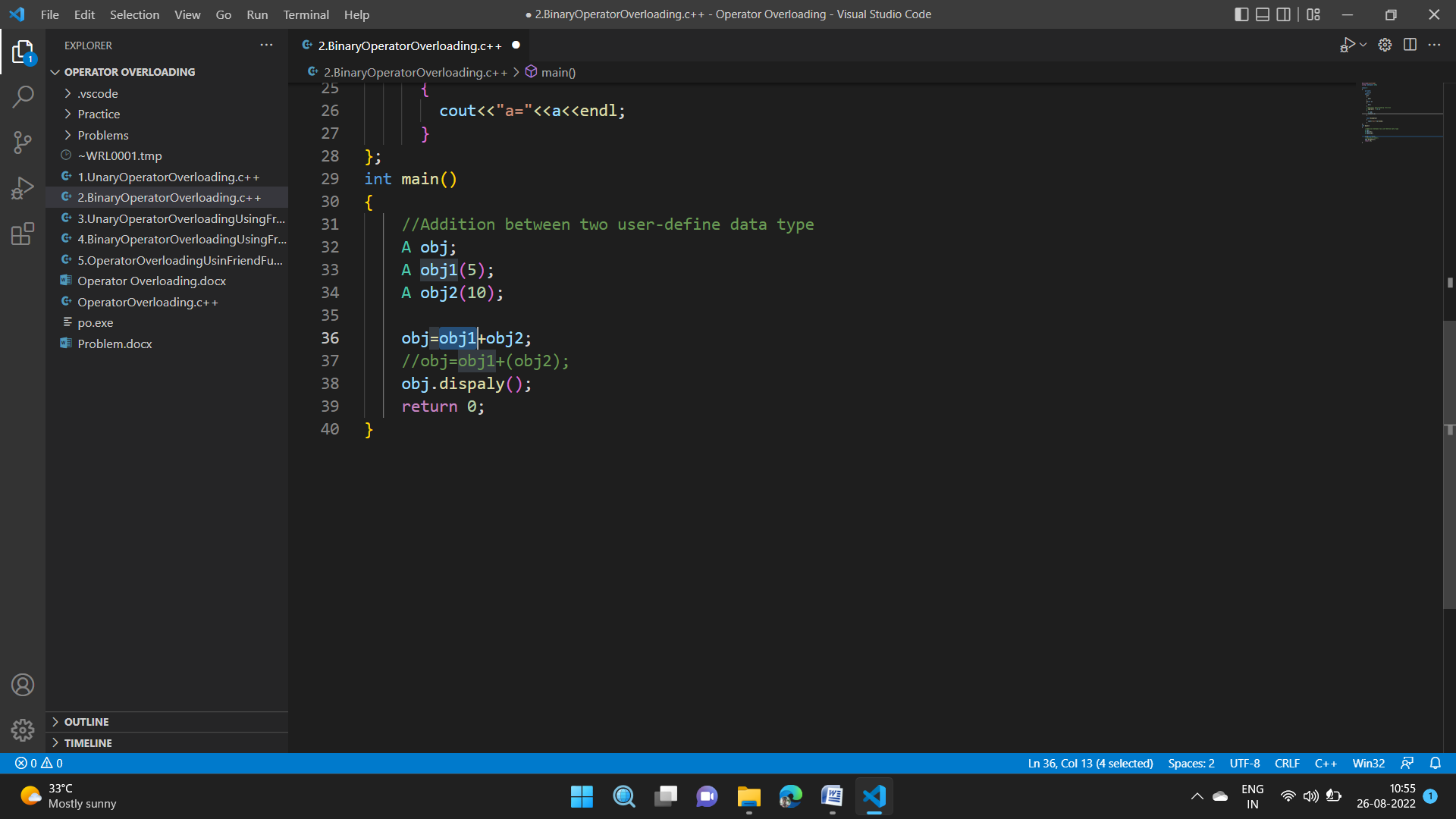
* **When unary operators are overloaded through a member function takes no explicit arguments.**





* **When binary operators are overloaded through a member function takes one explicit argument.**

****

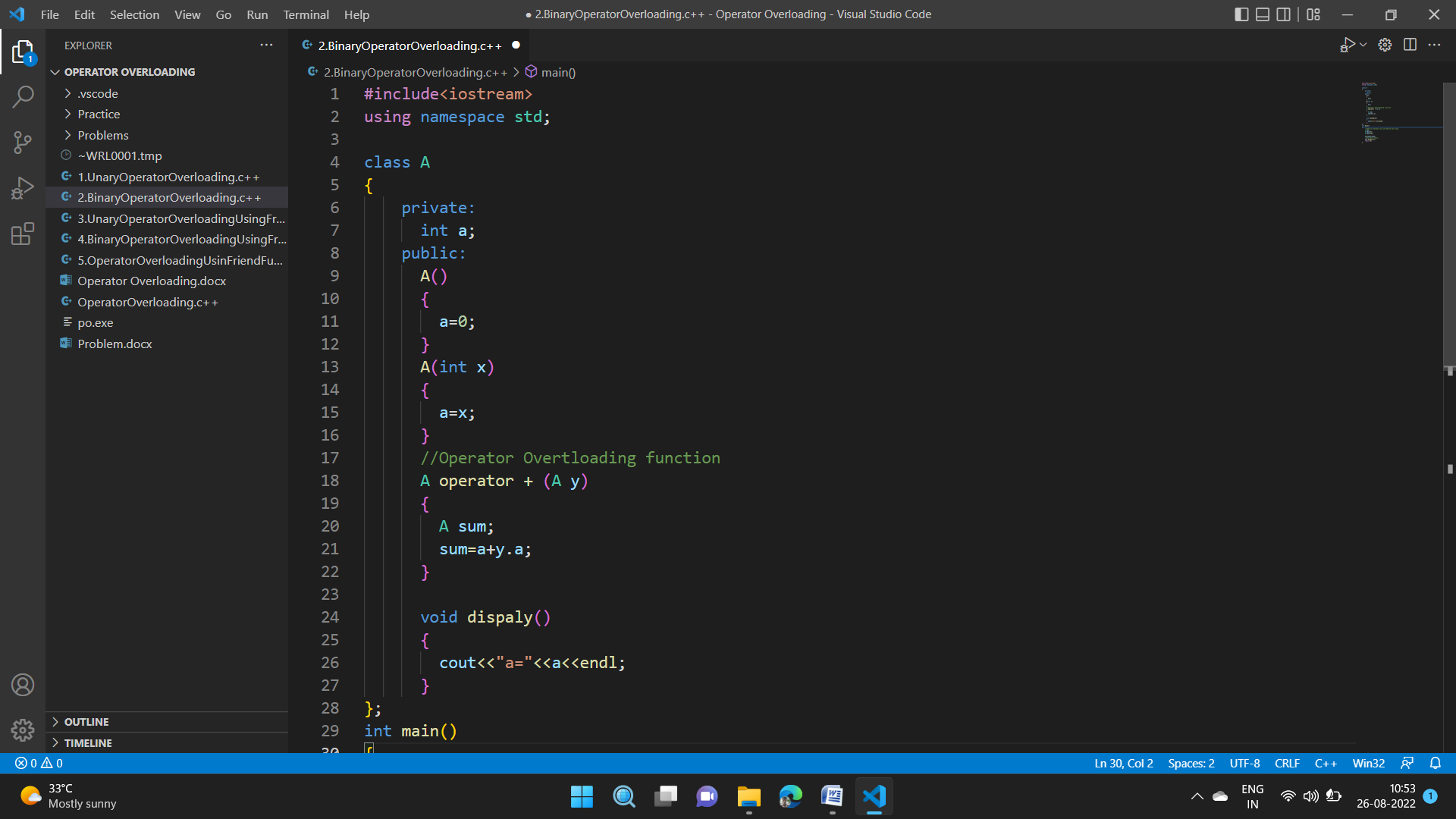
****

**obj=obj1+obj2; //invoke operator+() function**

* **In binary operator overloading using member function, the left hand side operand (obj1) is used to invoke the operator function and the right hand side operand (obj2) is passed as an argument.**
* **We know that a member function can be invoked only by an object of the same class. Here obj1 is used to invoke the operator function and the right hand side operand (obj2) is passed as an argument. The above invocation statement is equivalent to**

**obj=obj1.operator+(obj2);**

* **We can avoid the creation of the sum object by replacing the entire function body by the following statement:**

** return A(a+y.a); //invokes constructor2**

**What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object.**

**Using Friend Function:**

**Friend function may be used in the place of member function for operator overloading.**

**There are certain situations where we would like to use a friend function rather than a member function. For instance, consider a situation where we need to use two different types of operands for binary operator, say, one an object and another a built-in type as shown below,**

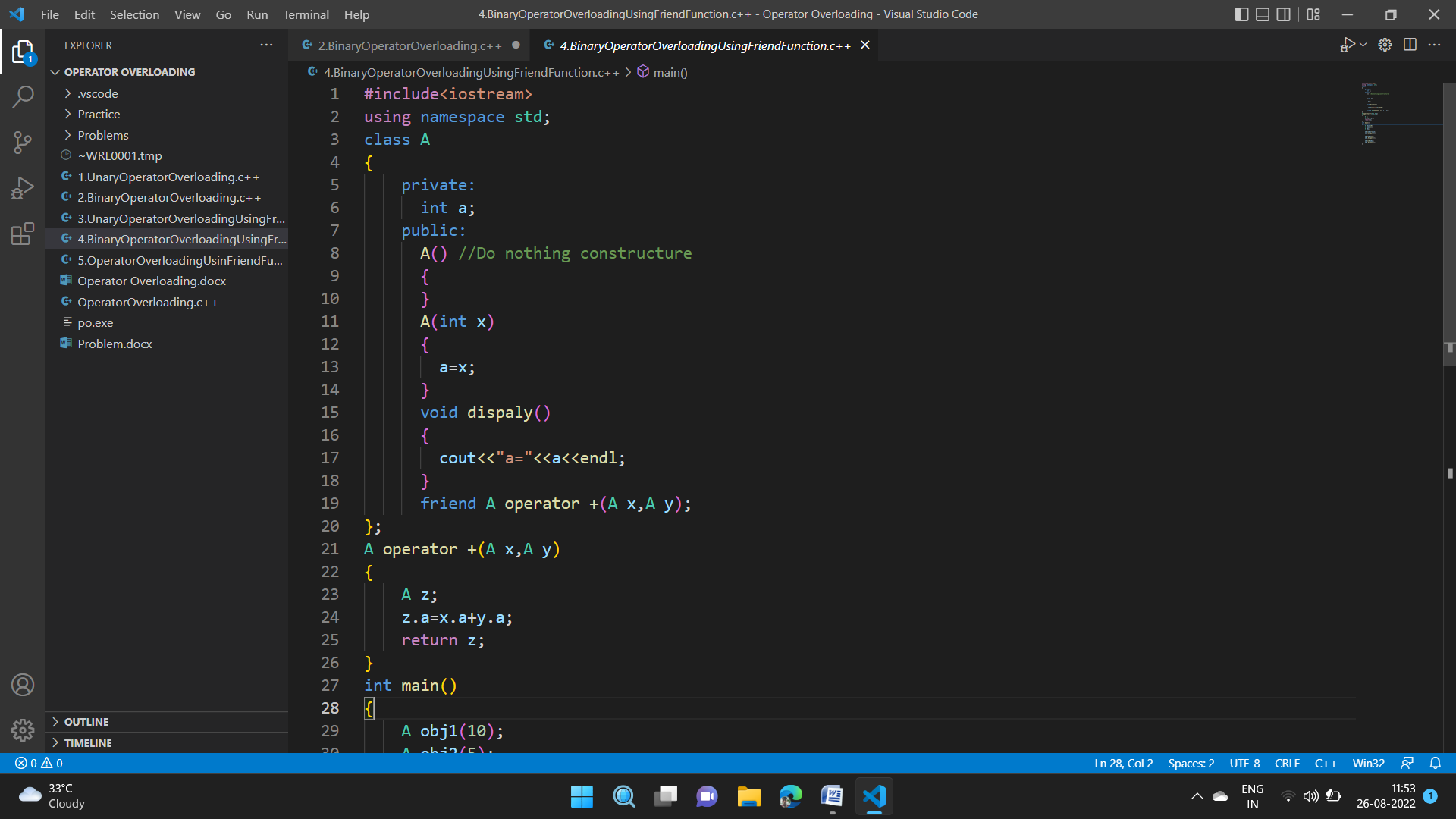
**A=B+2;**

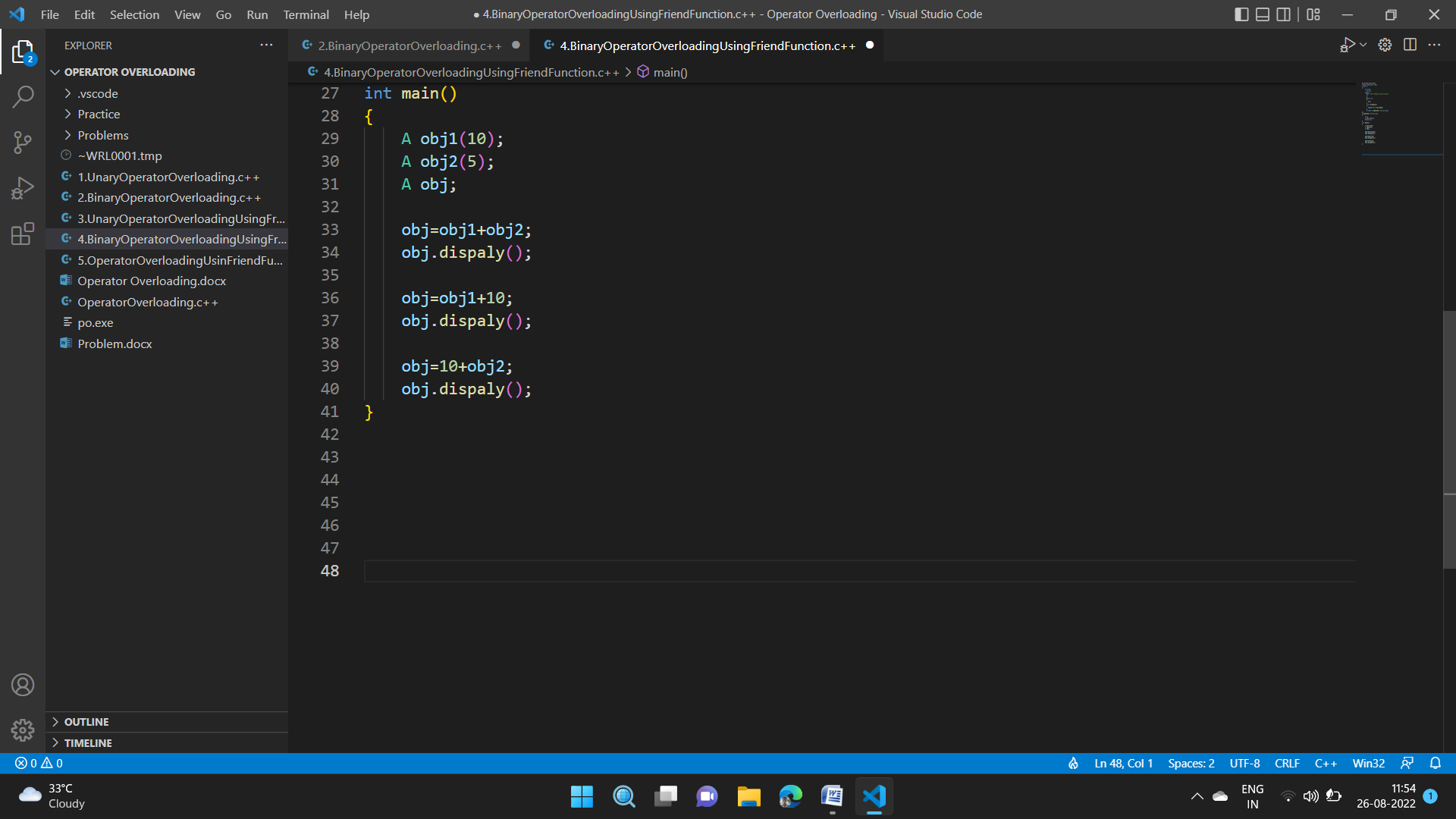
**where A and B are object of same class. This will work for member function but the statement**

**A=2+B;**

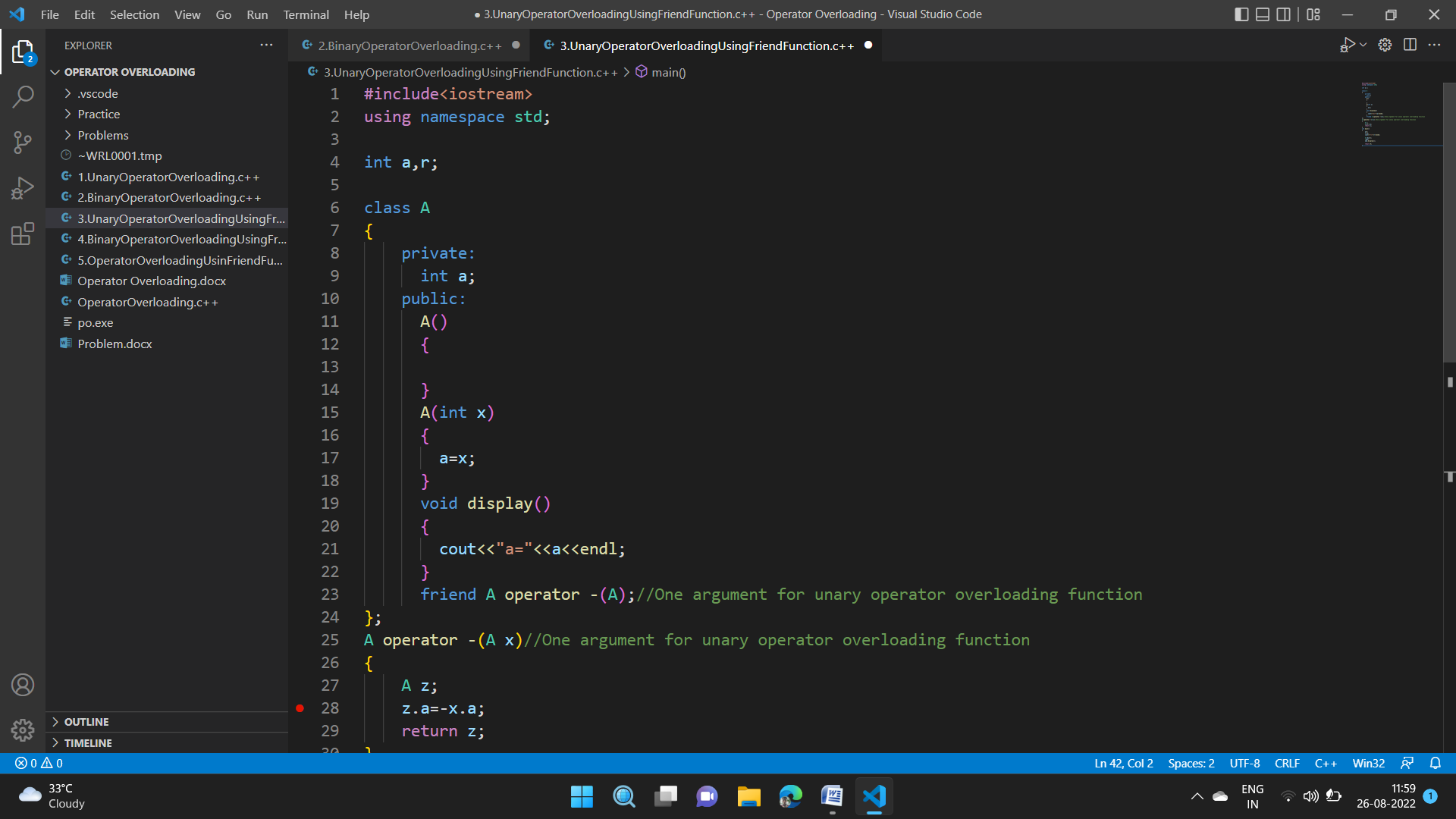
**will not work. This is because the left hand operand which responsible for calling the member functions should be an object of same class. However, friend function allows both approaches.**

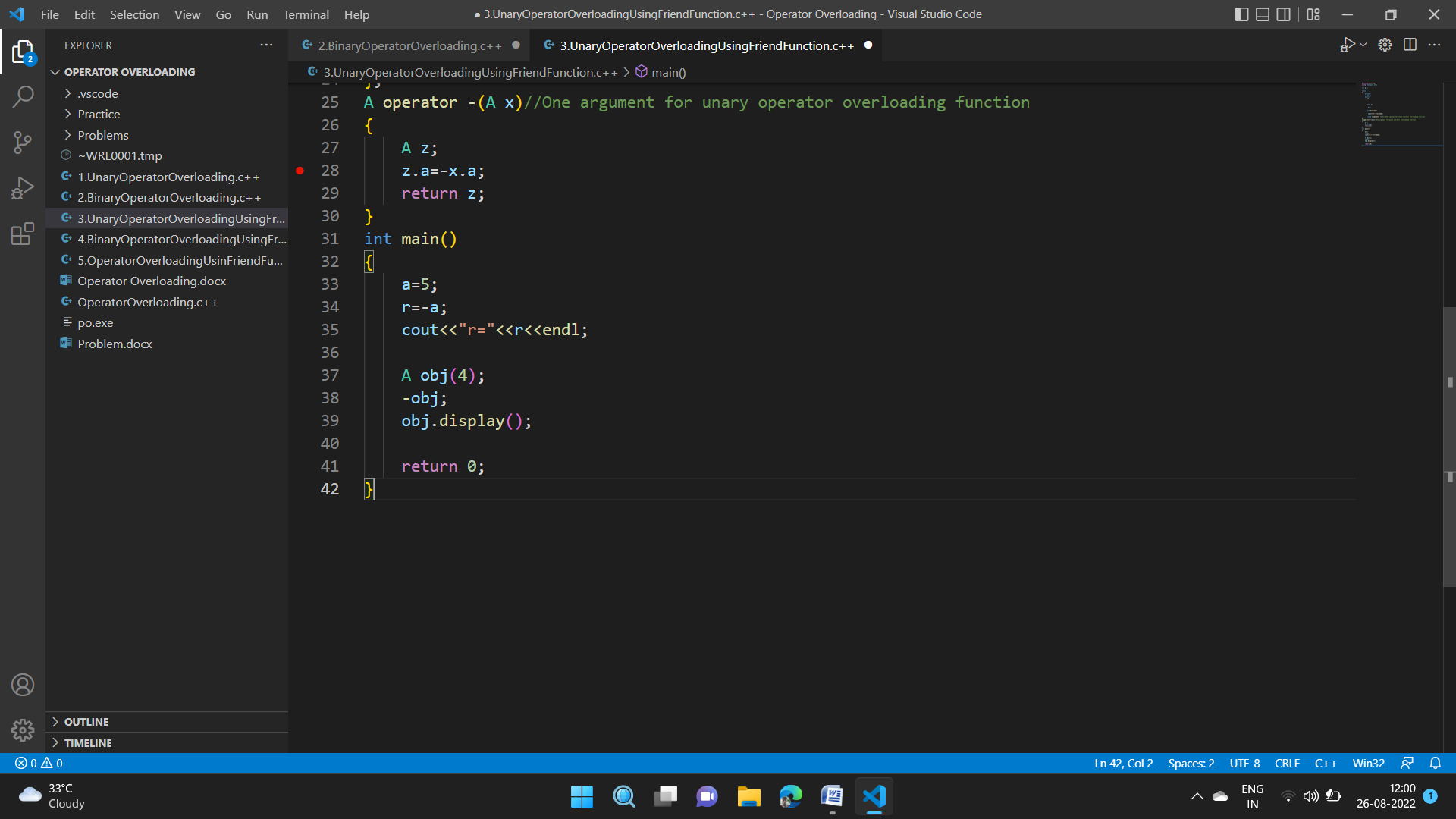
* **When binary operators are overloaded through a friend function takes two explicit arguments.**

****

****

* **When unary operators are overloaded through a friend function takes one explicit argument.**

****

****

**What is the difference between operator functions and normal functions?**   
**Operator functions are the same as normal functions. The only differences are that the name of an operator function is always the operator keyword followed by the symbol of the operator and operator functions are called when the corresponding operator is used.   
Following is an example of a global operator function.**

**Operators that can be overloaded:**

1. **Binary Arithmetic     ->     +, -, \*, /, %**
2. **Unary Arithmetic     ->     +, -, ++, —**
3. **Assignment     ->     =, +=,\*=, /=,-=, %=**
4. **Bit- wise      ->     & , | , << , >> , ~ , ^**
5. **De-referencing     ->     (->)**
6. **Dynamic memory allocation and De-allocation     ->     New, delete**
7. **Subscript     ->     [ ]**
8. **Function call     ->     ()**
9. **Logical      ->     &,  | |, !**

**10. Relational     ->     >, < , = =, <=, >=**

**Operators that cannot be overloaded:**

* **We cannot overload every operator in C++, there are some exceptions.**

**Exception for operator overloading are --**

* **Class member access operator ( . )**
* **Pointer to member operator ( .\* )**
* **Scope resolution operator( :: )**
* **Size operator ( sizeof() )**
* **Conditional operator ( ?: )**
* **Following operator cannot be used when a friend function is used to perform operator overloading:**
* **Assignment Operator ( = )**
* **Function call Operator ( () )**
* **Subscript Operator ( [] )**
* **Class member access operator ( -> )**

**Why can’t the above-stated operators be overloaded?**

**1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.**

**2. typeid: This provides a CPP program with the ability to recover the actual derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type ‘look’ like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.**

**3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.**

**4. Class member access operators (.(dot), .\* (pointer to member operator)): The importance and implicit use of class member access operators can be understood through the following example:**

**#include <iostream>**

**using namespace std;**

**class ComplexNumber{**

**private:**

**int real;**

**int imaginary;**

**public:**

**ComplexNumber(int real, int imaginary){**

**this->real = real;**

**this->imaginary = imaginary;**

**}**

**void print(){**

**cout<<real<<" + i"<<imaginary;**

**}**

**ComplexNumber operator+ (ComplexNumber c2){**

**ComplexNumber c3(0,0);**

**c3.real = this->real+c2.real;**

**c3.imaginary = this->imaginary + c2.imaginary;**

**return c3;**

**}**

**};**

**int main() {**

**ComplexNumber c1(3,5);**

**ComplexNumber c2(2,4);**

**ComplexNumber c3 = c1 + c2;**

**c3.print();**

**return 0;**

**}**

**Output**

**5 + i9**

**The statement ComplexNumber c3 = c1 + c2; is internally translated as ComplexNumber c3 = c1.operator+ (c2); in order to invoke the operator function. The argument c1 is implicitly passed using the ‘.’ operator. The next statement also makes use of the dot operator to access the member function print and pass c3 as an argument. Thus, in order to ensure a reliable and non-ambiguous system of accessing class members, the predefined mechanism using class member access operators is absolutely essential. Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.**

**5. Ternary or conditional (?:): The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.**

**conditional statement ? expression1 (if statement is TRUE) : expression2 (else)**

**A function overloading the ternary operator for a class say ABC using the definition**

**ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);**

**would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.**

**Restriction for Operator Overloading:**

* **Existing operators can only be overloaded, but the new operators cannot be overloaded.**
* **Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.**
* **Numbers of Operands cannot be changed. Unary operator remains unary, binary remains binary etc.**
* **Cannot redefine the meaning of a procedure. You cannot change how integers are added.**

**Importance Points of Operator Overloading:**

* **Two operators = and & are already overloaded by default in C++. Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behaviour is the same as the copy constructor).We do not need to create an operator function.**
* **We can also write conversion operators that can be used to convert one type to another type.**

**#include <iostream>**

**using namespace std;**

**class Fraction**

**{**

**private:**

**int num, den;**

**public:**

**Fraction(int n, int d) { num = n; den = d; }**

**// Conversion operator: return float value of fraction**

**operator float() const {**

**return float(num) / float(den);**

**};**

**}**

**int main() {**

**Fraction f(2, 5);**

**float val = f;**

cout << val << '\n';

return 0;

}

Output

0.4

**Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.**

* **Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.**

**include <iostream>**

**using namespace std;**

**class Point**

**{**

**private:**

**int x, y;**

**public:**

**Point(int i = 0, int j = 0) {**

**x = i;  y = j;**

**}**

**void print() {**

**cout << "x = " << x << ", y = " << y << '\n';**

**}**

**};**

**int main() {**

**Point t(20, 20);**

**t.print();**

**t = 30;   // Member x of t becomes 30**

**t.print();**

**return 0;**

**}**