CHAPTER-4

Operator Overloading

What is operator overloading?

Operator overloading is a method of making an operator work for user-defined classes, by giving it a special, user-defined meaning. This allows operators like +, -, *, etc., to perform custom operations on objects.

Operator overloading is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

For example:

The + operator is normally used to add built-in data types like int, float, or double. But through operator overloading, it can also be used to perform addition on user-defined types, like objects of a class.

Operator overloading (Contd..)

In **operator overloading**, we can change what an operator does for **user-defined types** like classes.
But we **cannot change** the basic rules of the operator.

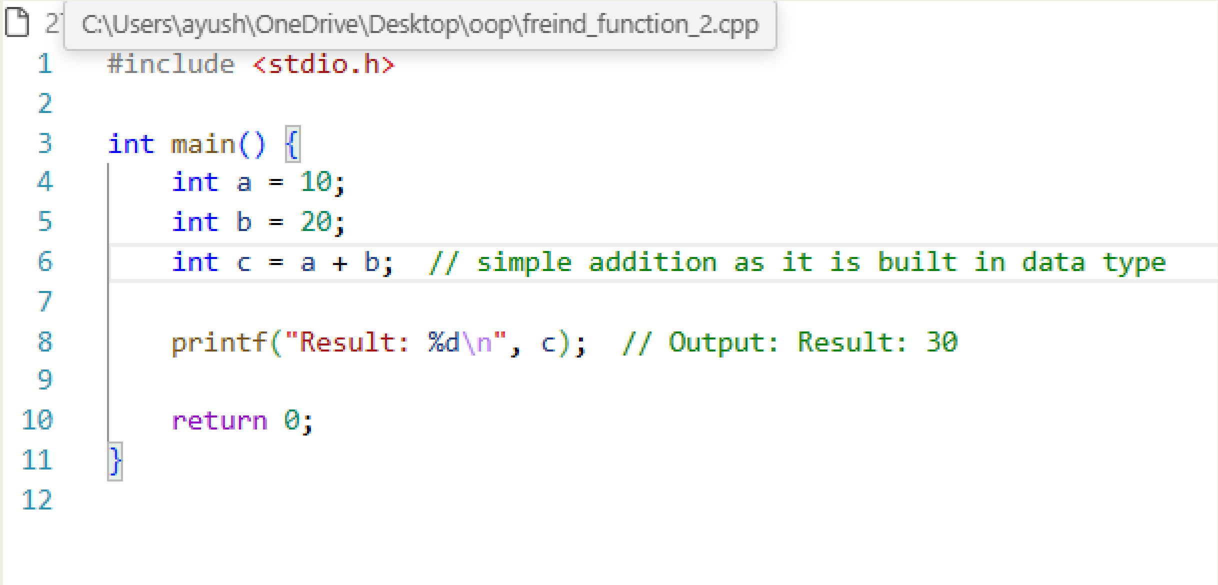
- This means:

- We **cannot change** the number of values (operands) it works with.
- We **cannot change** the **priority** (precedence) of the operator.
- We **cannot change** the **direction** (associativity) of how it's used.

We can overload all the c++ operator except

1. Class member access operator (. , *)
2. Scope resolution operator (: :)
3. Size operator (Sizeof())
4. Conditional operator (? :)

Before example of operator overload

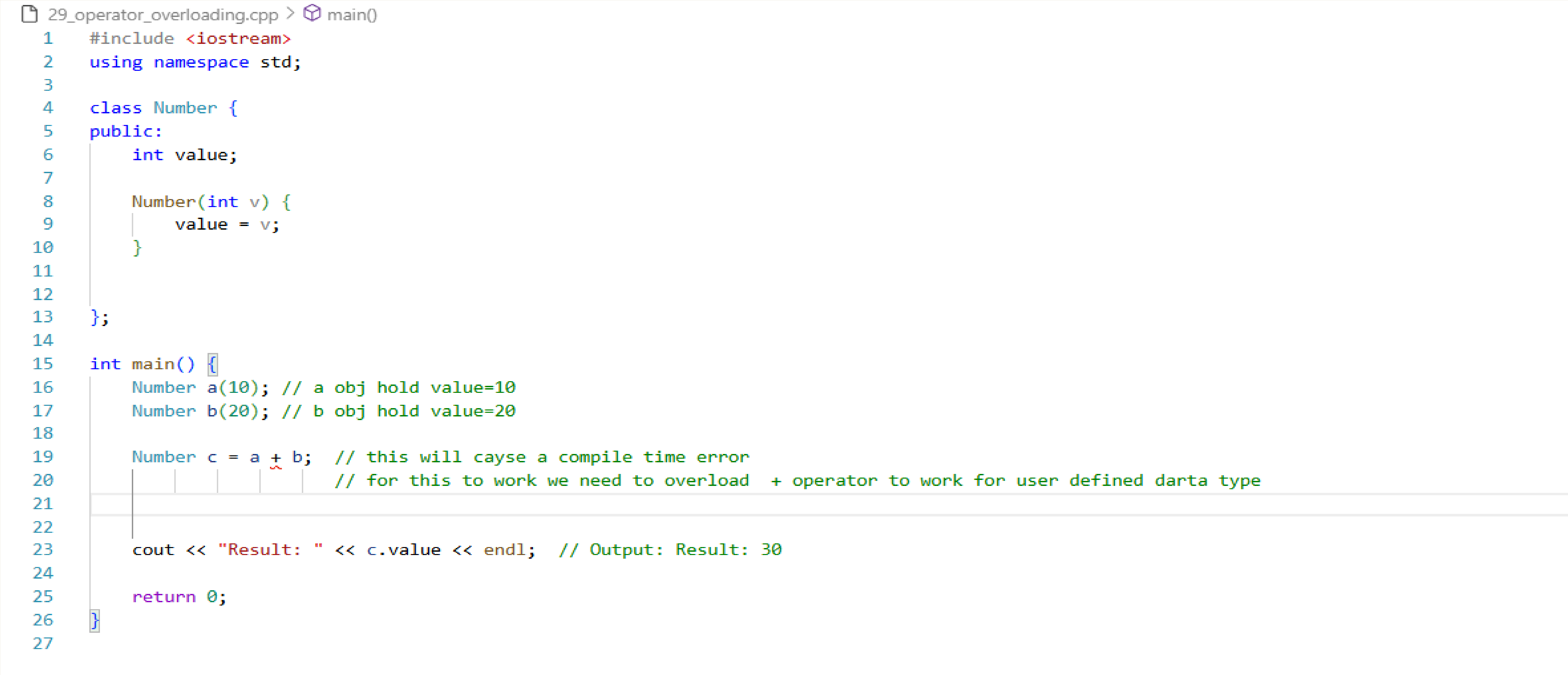


C:\Users\ayush\OneDrive\Desktop\oop\freind_function_2.cpp

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;
    int c = a + b;  // simple addition as it is built in data type

    printf("Result: %d\n", c);  // Output: Result: 30

    return 0;
}

```

Note: convert the above c program in to c++ while preparing a note

Before example of operator overload



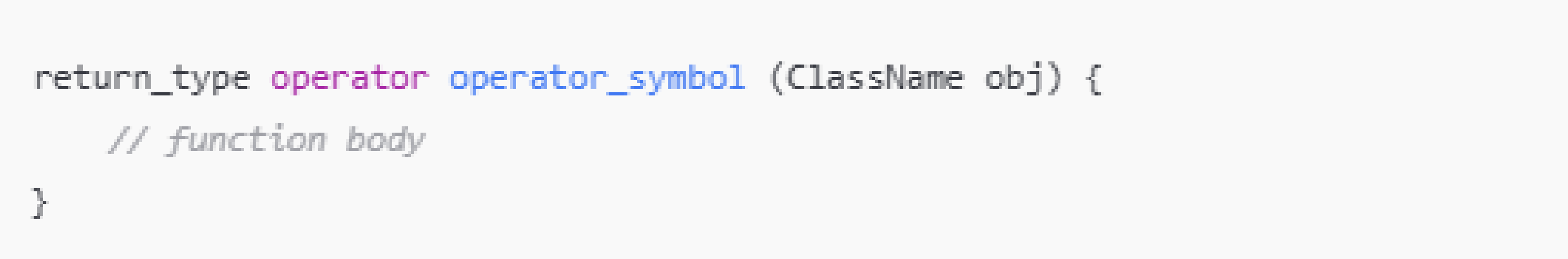
```cpp
#include <iostream>
using namespace std;

class Number {
public:
    int value;

    Number(int v) {
        value = v;
    }


};

int main() {
    Number a(10); // a obj hold value=10
    Number b(20); // b obj hold value=20

    Number c = a + b;  // this will cayse a compile time error
                       // for this to work we need to overload  + operator to work for user defined darta type


    cout << "Result: " << c.value << endl;  // Output: Result: 30

    return 0;
}

```

Contd..

This code will not compile because C++ doesn’t know how to add two objects of type Number.

The + operator works **for built-in types like int, but for custom classes**, we need to tell the compiler **how to add objects by overloading the + operator**.

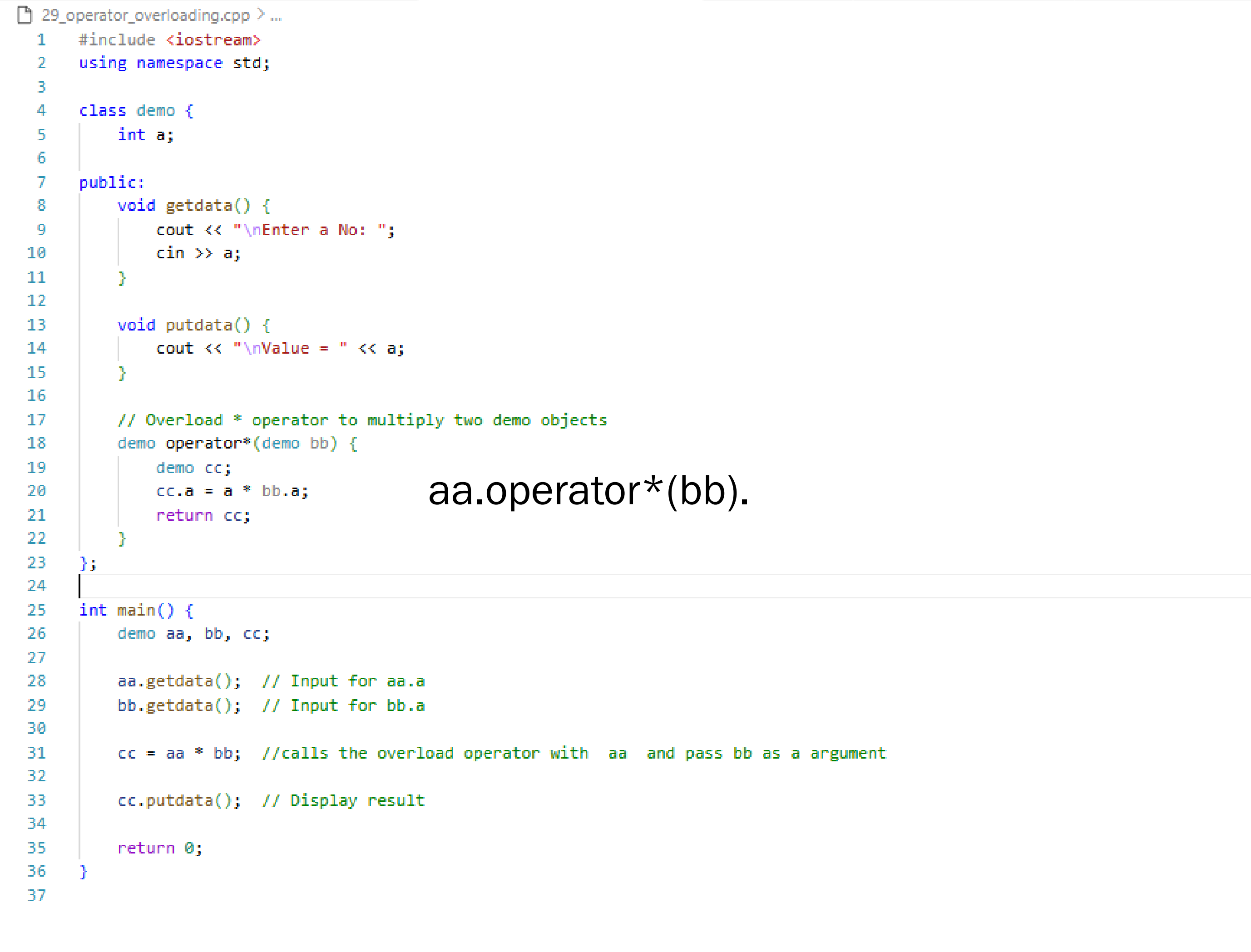
Syntax for defining operator overloading



```
return_type operator operator_symbol (ClassName obj) {
    // function body
}
```

- return_type: The type the operator returns — often the same class type.
- operator: The keyword to define operator overloading.
- operator_symbol: The operator you want to overload, for example: +, -, *, /, etc.
- (ClassName obj): The parameter, usually the right-hand side operand (passed by value or reference).

Syntax for defining operator overloading



29_operator_overloading.cpp > ...

```cpp
#include <iostream>
using namespace std;

class demo {
    int a;

public:
    void getdata() {
        cout << "\nEnter a No: ";
        cin >> a;
    }

    void putdata() {
        cout << "\nValue = " << a;
    }

    // Overload * operator to multiply two demo objects
    demo operator*(demo bb) {
        demo cc;
        cc.a = a * bb.a;
        return cc;
    }
};

int main() {
    demo aa, bb, cc;

    aa.getdata();  // Input for aa.a
    bb.getdata();  // Input for bb.a

    cc = aa * bb;  //calls the overload operator with  aa  and pass bb as a argument

    cc.putdata();  // Display result

    return 0;
}
```

aa.operator*(bb).

Here we are **calling the operator method with the object aa** and bb is passed as a argument to the operator

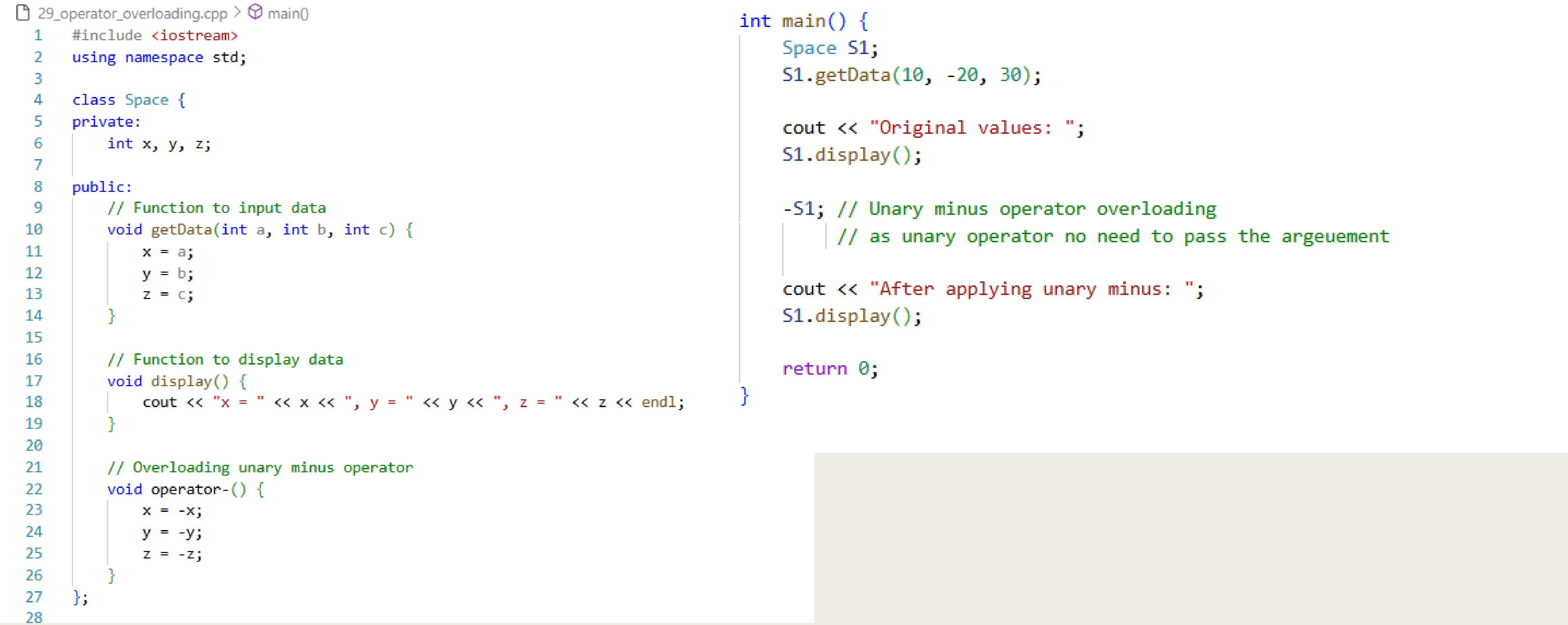
In **cc.a =a * bb.a // a simply means this.a**

Here when we directly write a it means this.a and it is a value of a object that is calling the method

Unary operator overloading

- Unary operator is an operator that works on only one operand (e.g., -a, ++a, --a).
- When we overload these unary operators in a class to define custom behavior, it is called unary operator overloading.

Unary operator overloading

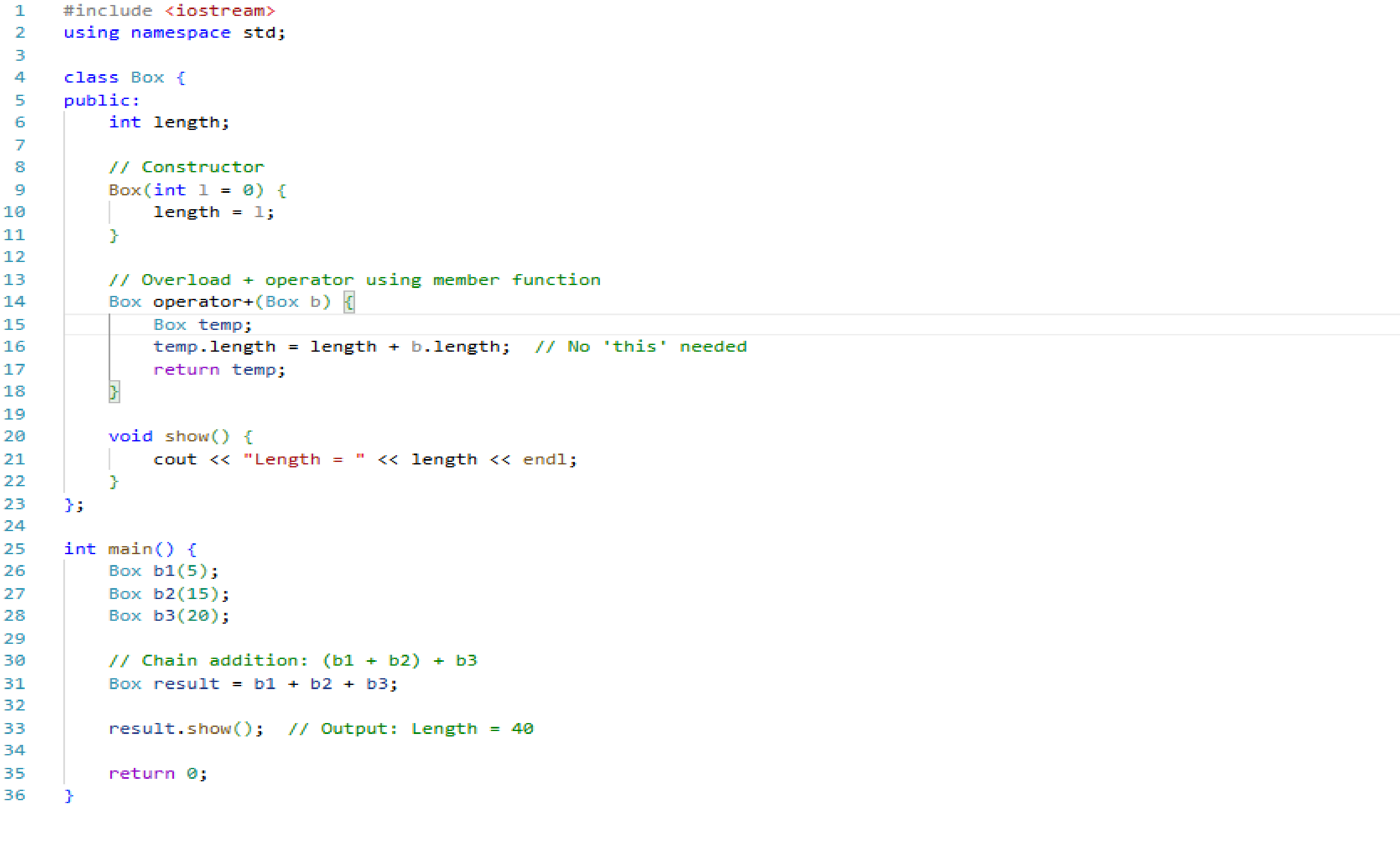


29_operator_overloading.cpp > main()

```cpp
#include <iostream>
using namespace std;

class Space {
private:
    int x, y, z;

public:
    // Function to input data
    void getData(int a, int b, int c) {
        x = a;
        y = b;
        z = c;
    }

    // Function to display data
    void display() {
        cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
    }

    // Overloading unary minus operator
    void operator-() {
        x = -x;
        y = -y;
        z = -z;
    }
};

```

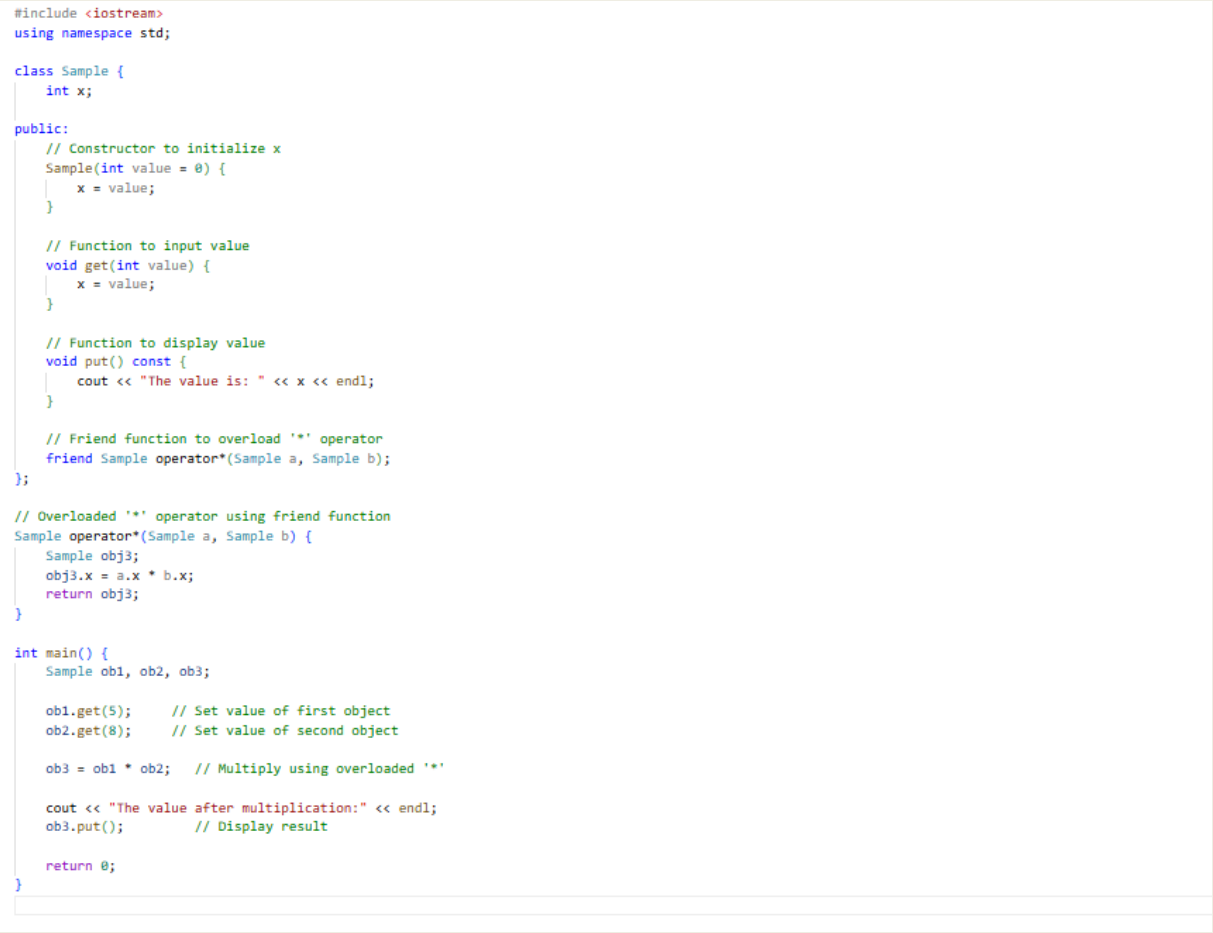
```cpp
int main() {
    Space S1;
    S1.getData(10, -20, 30);

    cout << "Original values: ";
    S1.display();

    -S1; // Unary minus operator overloading
        // as unary operator no need to pass the argeuement

    cout << "After applying unary minus: ";
    S1.display();

    return 0;
}
```

Operator overloading



```cpp
#include <iostream>
using namespace std;

class Box {
public:
    int length;

    // Constructor
    Box(int l = 0) {
        length = l;
    }

    // Overload + operator using member function
    Box operator+(Box b) {
        Box temp;
        temp.length = length + b.length;  // No 'this' needed
        return temp;
    }

    void show() {
        cout << "Length = " << length << endl;
    }
};

int main() {
    Box b1(5);
    Box b2(15);
    Box b3(20);

    // Chain addition: (b1 + b2) + b3
    Box result = b1 + b2 + b3;

    result.show();  // Output: Length = 40

    return 0;
}
```

Operator overloading with friend function



```cpp
#include <iostream>
using namespace std;

class Sample {
    int x;

public:
    // Constructor to initialize x
    Sample(int value = 0) {
        x = value;
    }

    // Function to input value
    void get(int value) {
        x = value;
    }

    // Function to display value
    void put() const {
        cout << "The value is: " << x << endl;
    }

    // Friend function to overload '*' operator
    friend Sample operator*(Sample a, Sample b);
};

// Overloaded '*' operator using friend function
Sample operator*(Sample a, Sample b) {
    Sample obj3;
    obj3.x = a.x * b.x;
    return obj3;
}

int main() {
    Sample ob1, ob2, ob3;

    ob1.get(5);     // Set value of first object
    ob2.get(8);     // Set value of second object

    ob3 = ob1 * ob2;   // Multiply using overloaded '*'

    cout << "The value after multiplication:" << endl;
    ob3.put();         // Display result

    return 0;
}
```

Here both a and b is passed as a parameter because we **cannot call the operator with the object of class** like previous

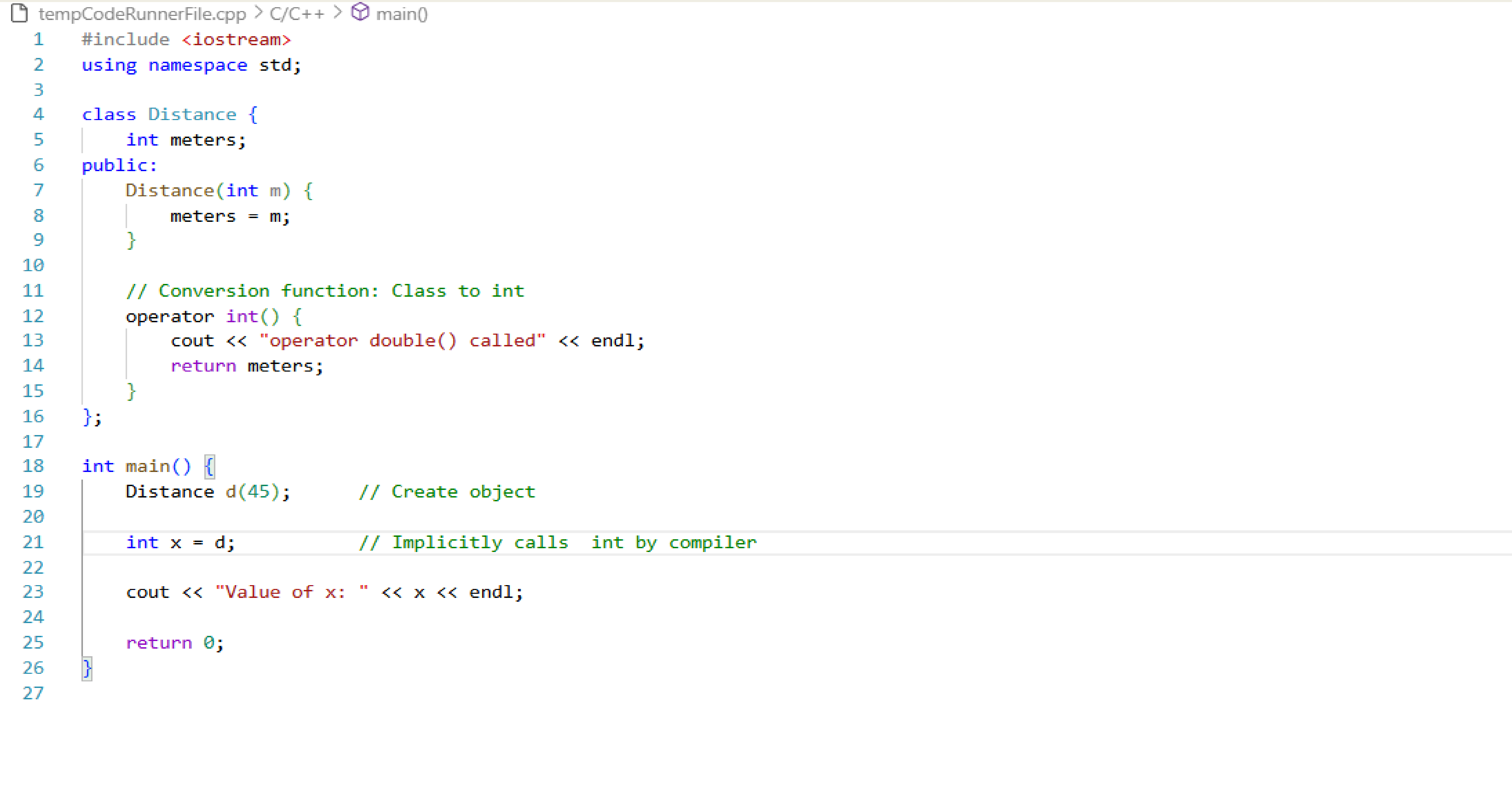
Type conversion

The process of converting one data type into another is called as a type conversion or a type casting

It is discussed already in detail in chapter2

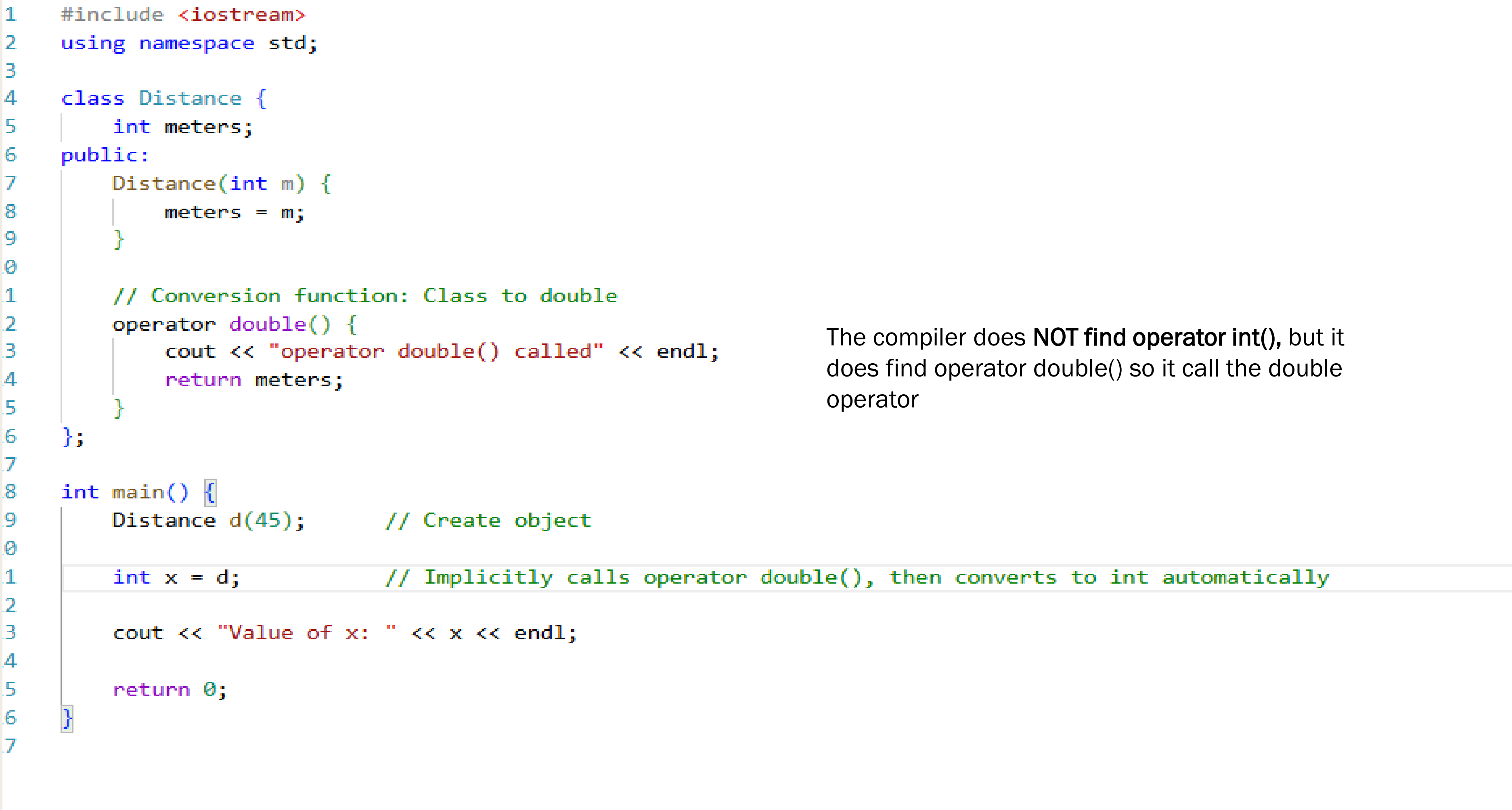
We are going to discuss here only about the situation that might arise in the data conversion between incompatible type which are as bellows

Conversion from class type to basic type



```cpp
#include <iostream>
using namespace std;

class Distance {
    int meters;
public:
    Distance(int m) {
        meters = m;
    }

    // Conversion function: Class to int
    operator int() {
        cout << "operator double() called" << endl;
        return meters;
    }
};

int main() {
    Distance d(45);      // Create object

    int x = d;           // Implicitly calls  int by compiler

    cout << "Value of x: " << x << endl;

    return 0;
}
```

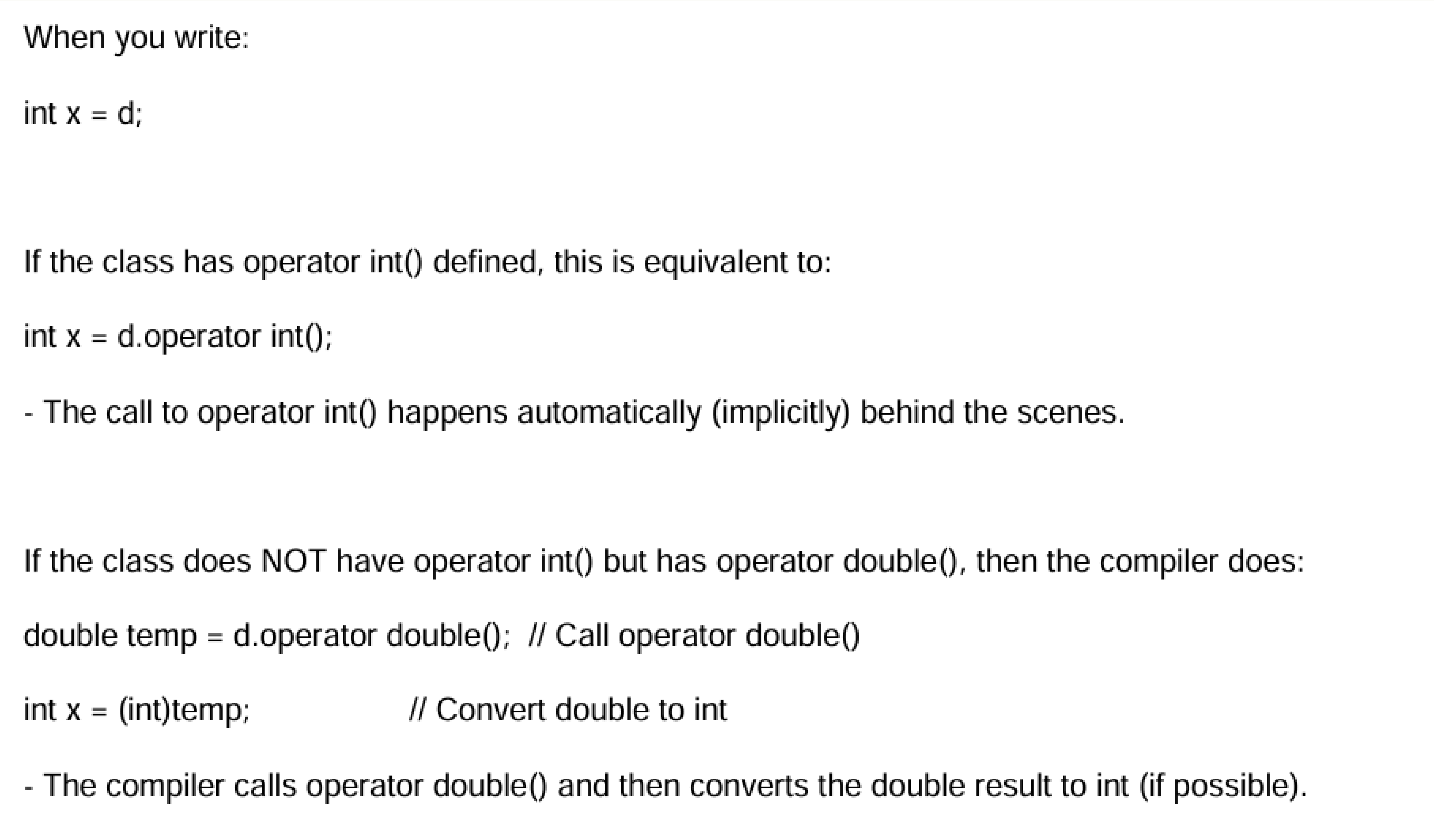
Be carefully



```cpp
#include <iostream>
using namespace std;

class Distance {
    int meters;
public:
    Distance(int m) {
        meters = m;
    }

    // Conversion function: Class to double
    operator double() {
        cout << "operator double() called" << endl;
        return meters;
    }
};

int main() {
    Distance d(45);      // Create object

    int x = d;           // Implicitly calls operator double(), then converts to int automatically

    cout << "Value of x: " << x << endl;

    return 0;
}
```

The compiler does **NOT find operator int(),** but it does find operator double() so it call the double operator

CONCEPT



When you write:

```
int x = d;
```

If the class has operator int() defined, this is equivalent to:

```
int x = d.operator int();
```

- The call to operator int() happens automatically (implicitly) behind the scenes.

If the class does NOT have operator int() but has operator double(), then the compiler does:

```
double temp = d.operator double();  // Call operator double()
int x = (int)temp;                   // Convert double to int
```

- The compiler calls operator double() and then converts the double result to int (if possible).

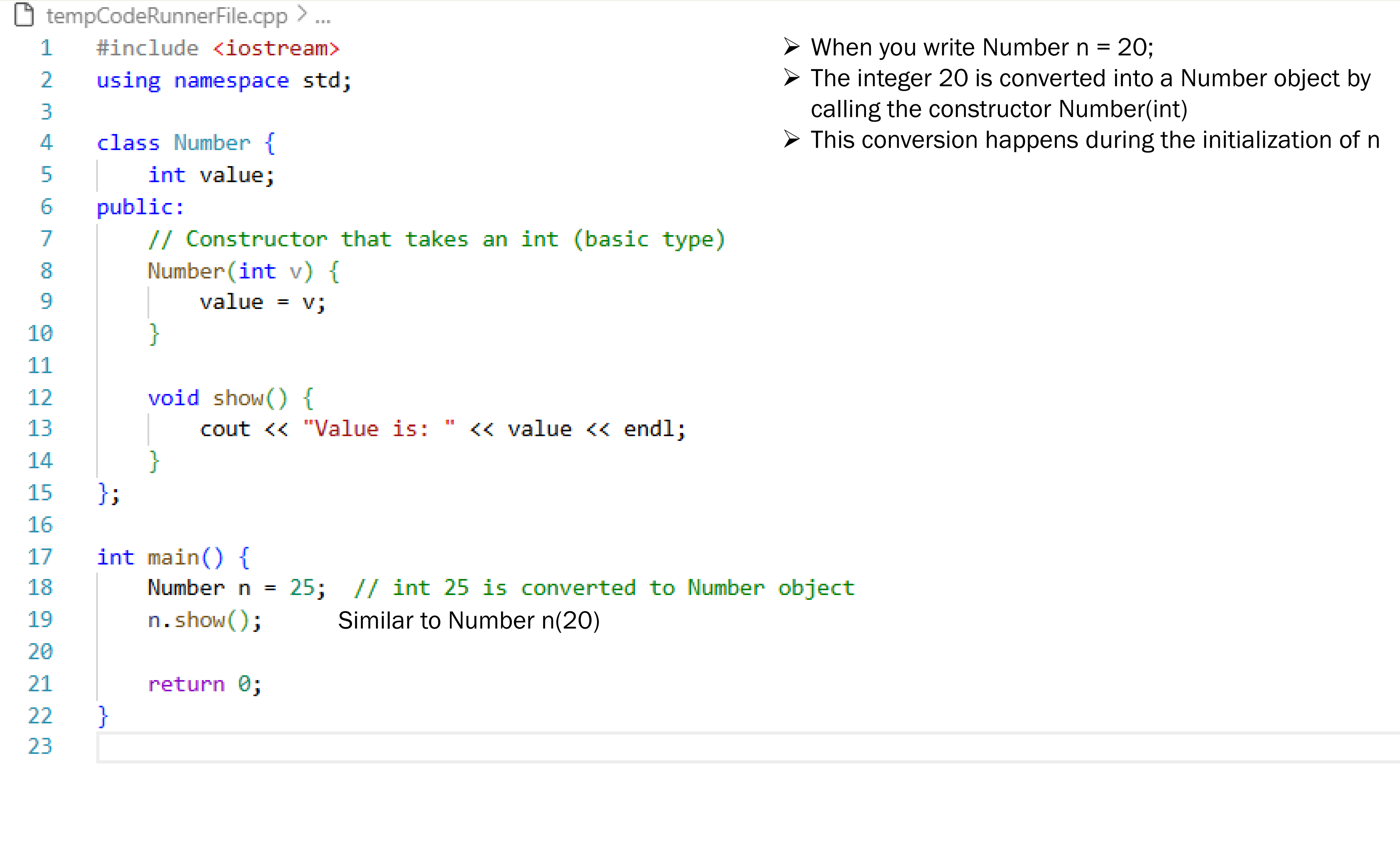
Type conversion

The process of converting one data type into another is called as a type conversion or a type casting

It is discussed already in detail in chapter2

We are going to discuss here only about the situation that might arise in the data conversion between incompatible type which are as bellows

Conversion of basic type to class type



```cpp
#include <iostream>
using namespace std;

class Number {
    int value;
public:
    // Constructor that takes an int (basic type)
    Number(int v) {
        value = v;
    }

    void show() {
        cout << "Value is: " << value << endl;
    }
};

int main() {
    Number n = 25;  // int 25 is converted to Number object
    n.show();

    return 0;
}

```

Similar to Number n(20)

- When you write Number n = 20;
- The integer 20 is converted into a Number object by calling the constructor Number(int)
- This conversion happens during the initialization of n

Conversion from One class type to another class type

Try Yourself Assignment

Consider a two class one rupees and another dollar and convert