Lesson 9

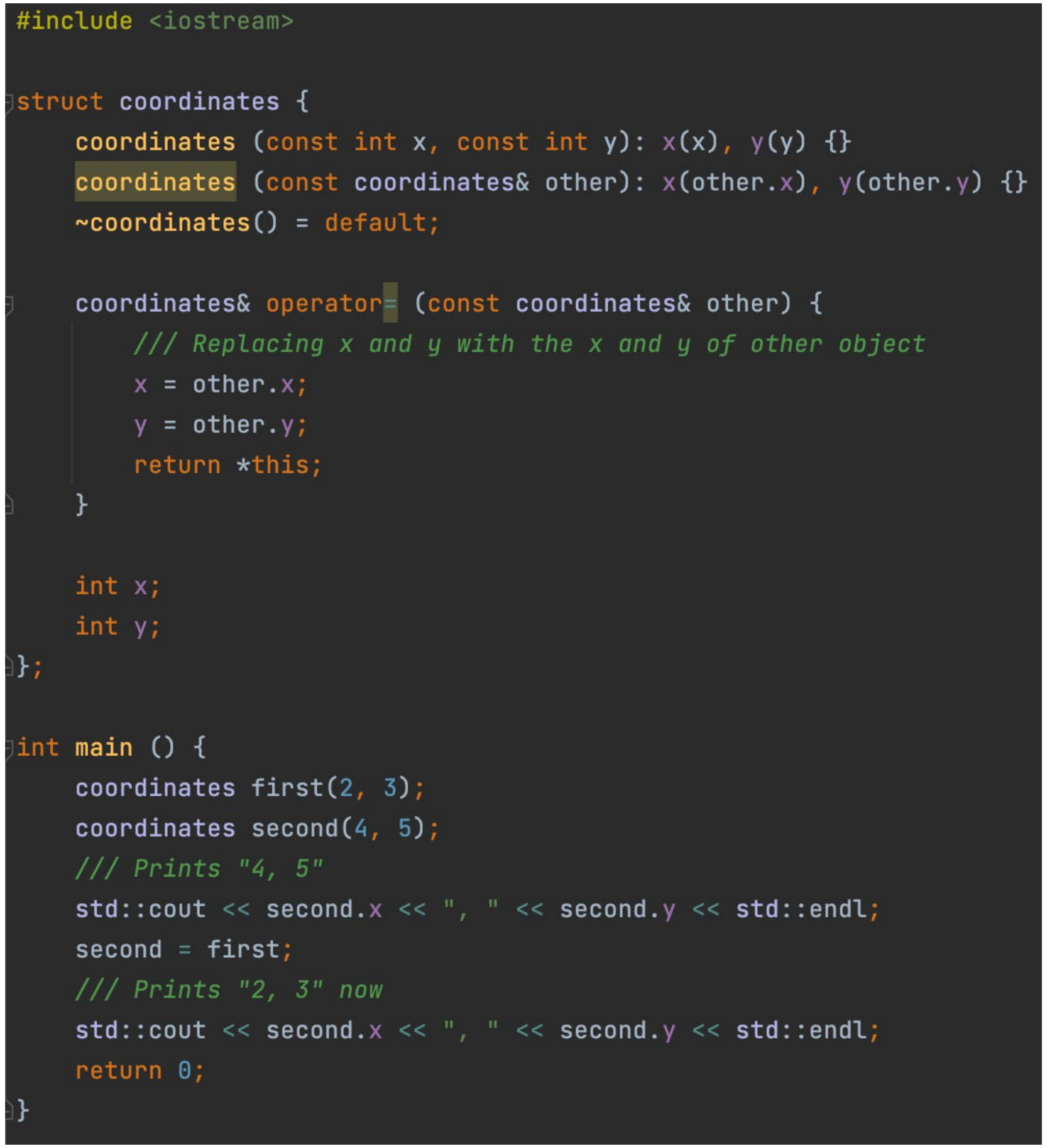
Classes and structures

Copy assignment operator

The copy assignment operator allows you to change the content of an already created object by copying the state of another object into it. The difference between the copy assignment operator and the copy constructor is that the copy assignment operator does NOT create a new object, but operates with existing ones.

Defining custom copy assignment operator:

Defining your own copy assignment operator for a class is very similar to defining a method.



```cpp
#include <iostream>

struct coordinates {
    coordinates (const int x, const int y): x(x), y(y) {}
    coordinates (const coordinates& other): x(other.x), y(other.y) {}
    ~coordinates() = default;

    coordinates& operator= (const coordinates& other) {
        /// Replacing x and y with the x and y of other object
        x = other.x;
        y = other.y;
        return *this;
    }

    int x;
    int y;
};

int main () {
    coordinates first(2, 3);
    coordinates second(4, 5);
    /// Prints "4, 5"
    std::cout << second.x << ", " << second.y << std::endl;
    second = first;
    /// Prints "2, 3" now
    std::cout << second.x << ", " << second.y << std::endl;
    return 0;
}
```

Operator= return type

Why operator= does not return void

Let's remember the very first lectures, namely, the lecture about expressions. In it, I talked about expressions of this type:
x = y = z;

For such expressions to work correctly, the result of the expression y = z (which is getting calculated first, of course) must be the value of the object obtained after the assignment.

Why operator= returns a reference

Because in some cases, we need a reference to the object we just assigned to!
(x = y) = z;

In this example, there will be two calls to the copy assignment operator:
x = y, and then x = z

Rule of three, recap

Rule of 3: If a class requires a user-defined **destructor**, a user-defined **copy constructor**, or a user-defined **copy assignment operator**, it almost certainly requires all three.

STL Containers, first cursory overview

Let's look at a few examples of classes built into the standard library that will be useful during homework and exam contest: **std::vector**, **std::array**

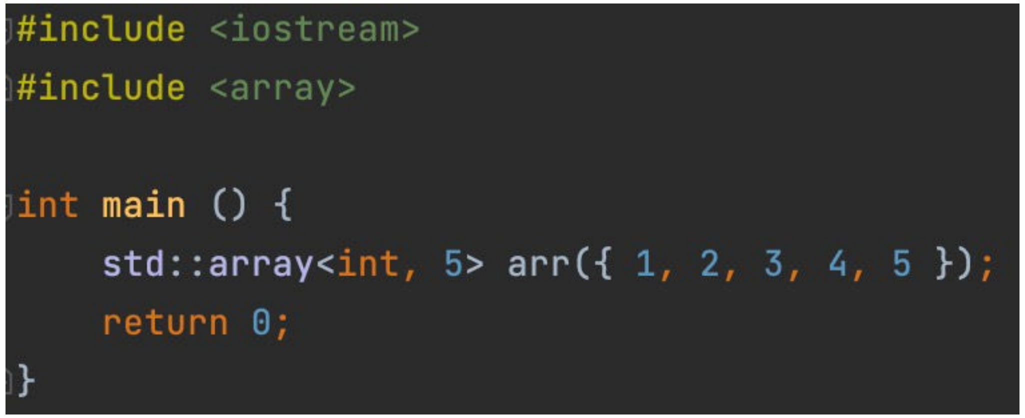
std::array and **std::vector** are called containers, they provide some storage for collection (sequence) of objects, or primitives. There are a lot of containers, we will all look at all of them in detail in the second semester.

std::array: constructor

std::array is a fixed, non-expandable array. Roughly speaking, it is analogous to C-style-array (for example, **int[]**). Array owns elements like any container, so we can say that it is using the RAII idiom.

To create an array, you need:

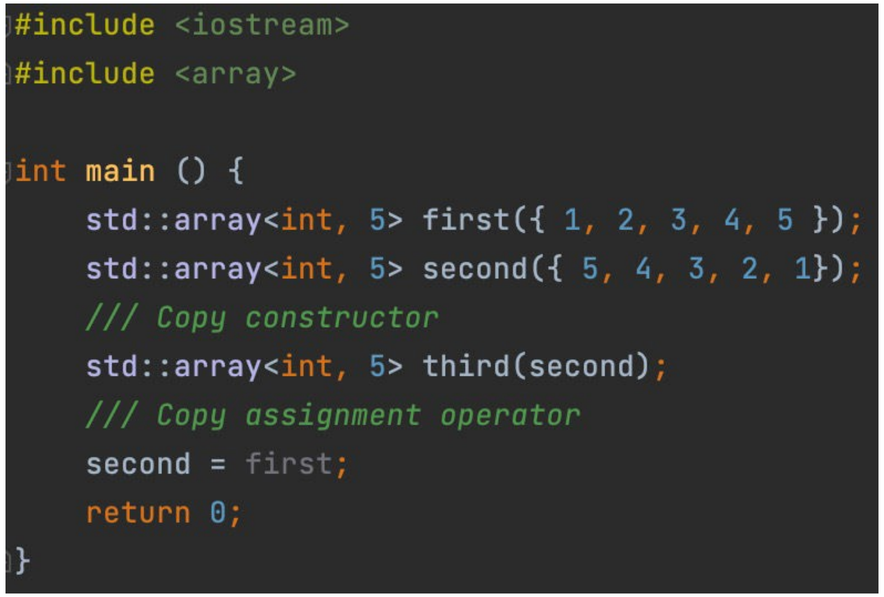
- To include <array> header
- To pass the type of the array element and the number of elements in the array in triangle brackets (this is called template parameters, we'll talk about how it works later)
- To call constructor from **std::initializer_list**



```
#include <iostream>
#include <array>

int main () {
    std::array<int, 5> arr({ 1, 2, 3, 4, 5 });
    return 0;
}
```

std::array: copy constructor and copy assignment operator

Naturally, the array supports both the copy constructor and the copy assignment operator

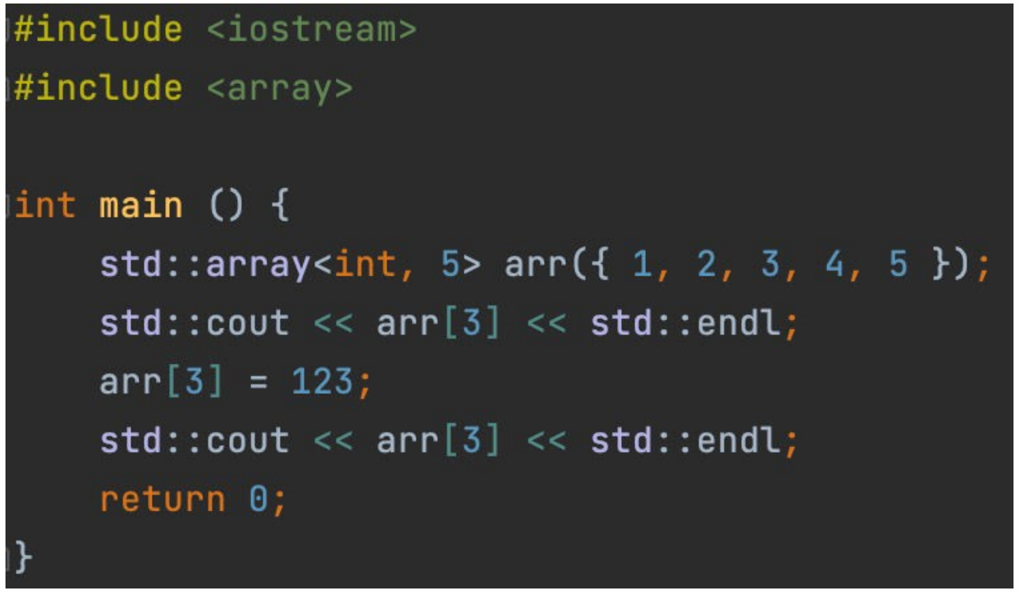


```cpp
#include <iostream>
#include <array>

int main () {
    std::array<int, 5> first({ 1, 2, 3, 4, 5 });
    std::array<int, 5> second({ 5, 4, 3, 2, 1});
    /// Copy constructor
    std::array<int, 5> third(second);
    /// Copy assignment operator
    second = first;
    return 0;
}
```

Of course, the types and sizes of the arrays must match for copying to be possible!

std::array: operator[]

You can use the operator[] to access array elements, just like a regular C-style array. This works because the operator[] is defined for the std::array class. Actually, it's better to say that it's REdefined.

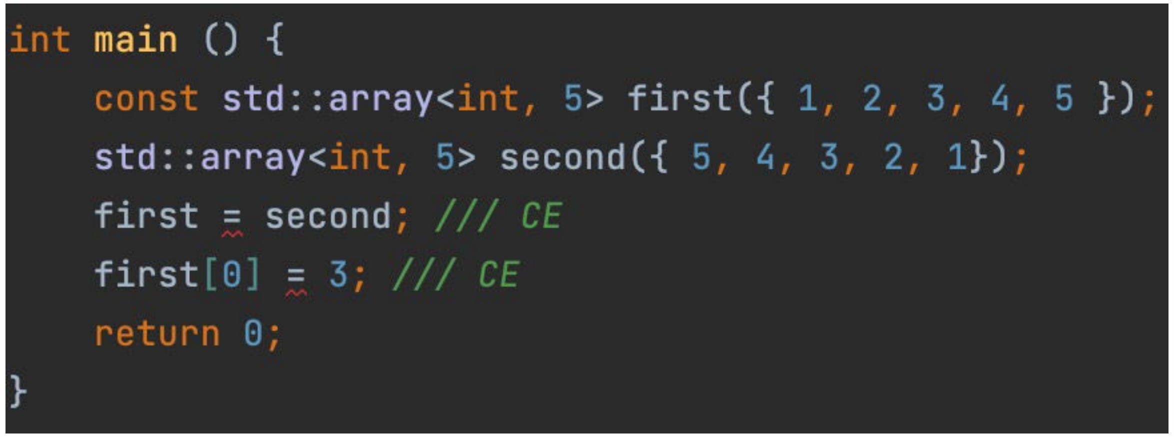


```
#include <iostream>
#include <array>

int main () {
    std::array<int, 5> arr({ 1, 2, 3, 4, 5 });
    std::cout << arr[3] << std::endl;
    arr[3] = 123;
    std::cout << arr[3] << std::endl;
    return 0;
}
```

The operator[] returns a reference to the i-th element of the array, so we can assign to this reference to change the value of the i-th element, meaning, that the operator[] provides both read access and write access to a specific element. This concept is called an accessor.

std::array: const qualifier

If we add a const qualifier to the array, we disallow any modifying operations on it:



```cpp
int main () {
    const std::array<int, 5> first({ 1, 2, 3, 4, 5 });
    std::array<int, 5> second({ 5, 4, 3, 2, 1});
    first = second; /// CE
    first[0] = 3; /// CE
    return 0;
}
```

in a const-qualified array, you cannot copy to it, using copy assignment operator and modify its elements through the operator[].

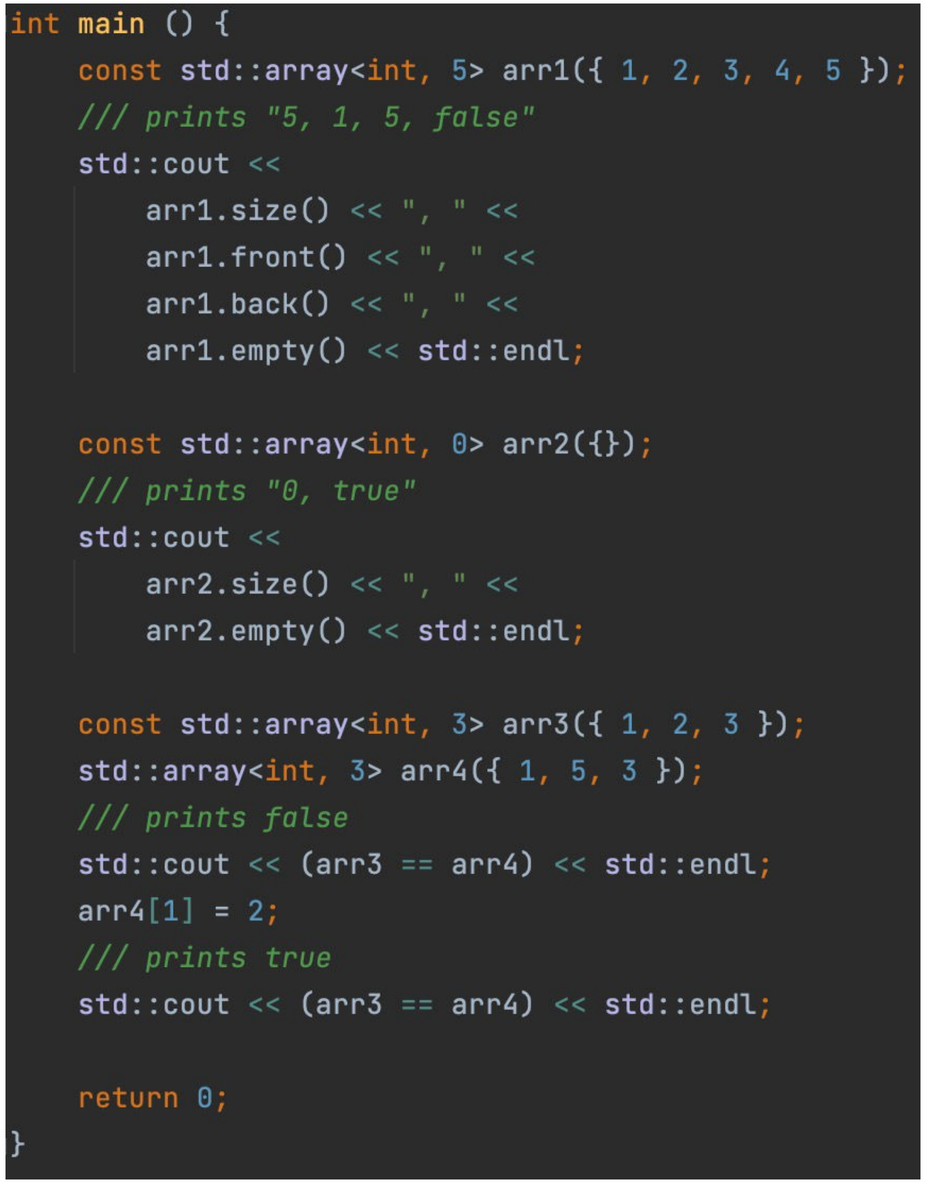
This is achieved by the fact that the operator[] has two versions: one for a non-const-qualified array, and the other for a const-qualified array.

std::array: another useful methods

- size() returns the amount of elements in the array
- front() is identical to the array[0]
- back() is identical to the array[array.size() - 1]
- empty() returns **true** if array is empty, **false** otherwise
- operator== compares two arrays

size() method returns **size_t** type, which is an **unsigned** variable, size of which is not less than **2 bytes**.

size_t can store the maximum size of a theoretically possible object of any type (including array).



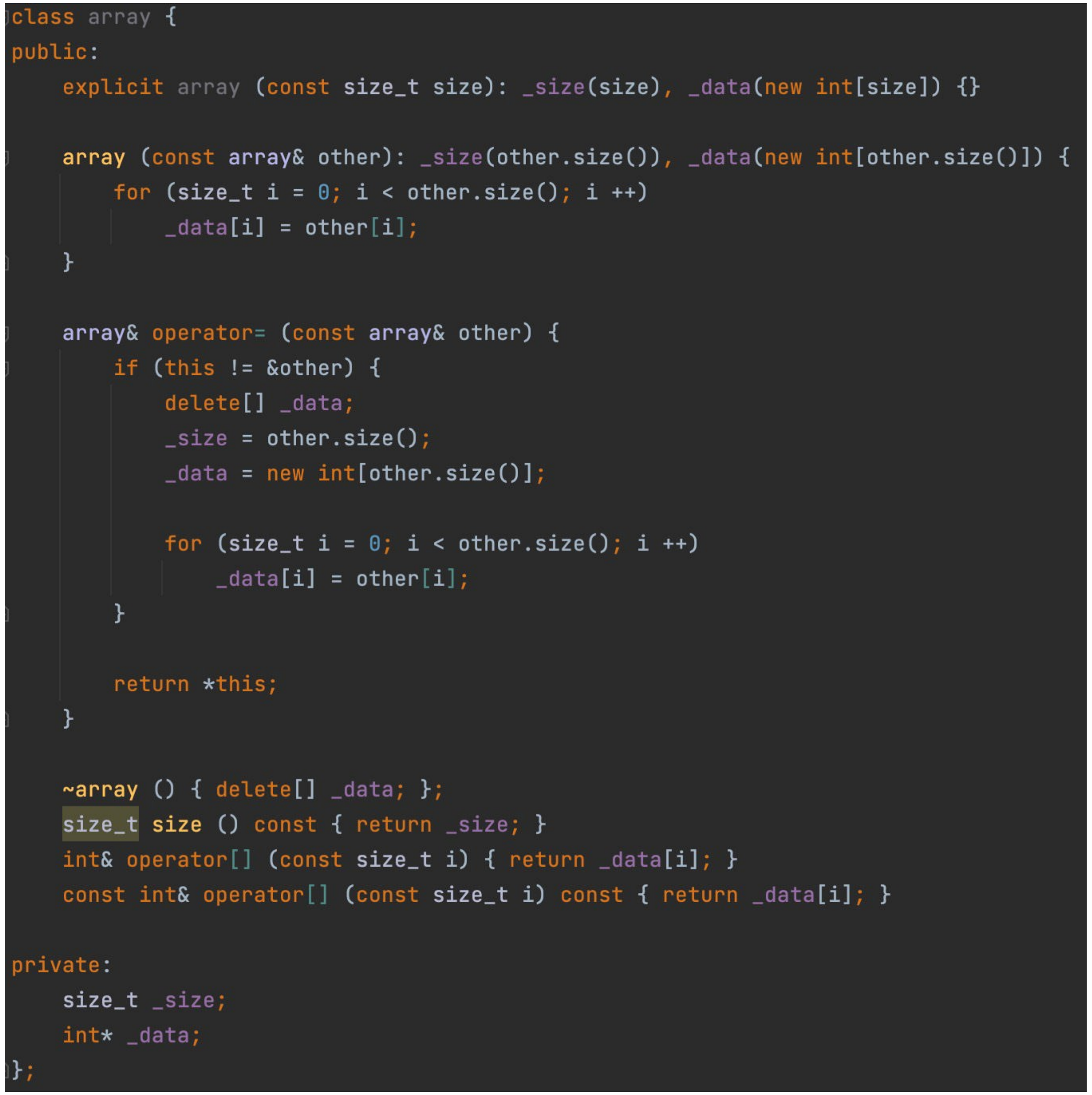
```cpp
int main () {
    const std::array<int, 5> arr1({ 1, 2, 3, 4, 5 });
    /// prints "5, 1, 5, false"
    std::cout <<
        arr1.size() << ", " <<
        arr1.front() << ", " <<
        arr1.back() << ", " <<
        arr1.empty() << std::endl;

    const std::array<int, 0> arr2({});
    /// prints "0, true"
    std::cout <<
        arr2.size() << ", " <<
        arr2.empty() << std::endl;

    const std::array<int, 3> arr3({ 1, 2, 3 });
    std::array<int, 3> arr4({ 1, 5, 3 });
    /// prints false
    std::cout << (arr3 == arr4) << std::endl;
    arr4[1] = 2;
    /// prints true
    std::cout << (arr3 == arr4) << std::endl;

    return 0;
}
```

std::array implementation

We will talk about **std::vector** next time, and today we will implement the non-template version of **std::array**

Key points:

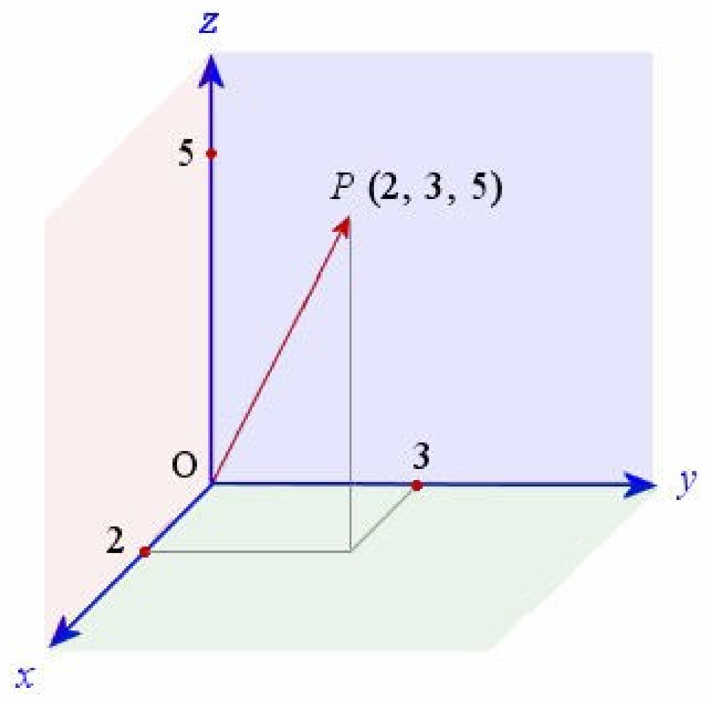
- allocate data in constructor, deallocate in destructor (RAII)
- operator= implementation: don’t forget to clear the state before copying
- operator= implementation: comparison with &other to avoid copying the object to itself
- two differently qualified operators[]: getter and setter
- size() method is a getter, it provides only read access to a certain property of an object, thus, it’s const qualified



```
class array {
public:
    explicit array (const size_t size): _size(size), _data(new int[size]) {}

    array (const array& other): _size(other.size()), _data(new int[other.size()]) {
        for (size_t i = 0; i < other.size(); i ++)
            _data[i] = other[i];
    }

    array& operator= (const array& other) {
        if (this != &other) {
            delete[] _data;
            _size = other.size();
            _data = new int[other.size()];

            for (size_t i = 0; i < other.size(); i ++)
                _data[i] = other[i];
        }

        return *this;
    }

    ~array () { delete[] _data; };
    size_t size () const { return _size; }
    int& operator[] (const size_t i) { return _data[i]; }
    const int& operator[] (const size_t i) const { return _data[i]; }

private:
    size_t _size;
    int* _data;
};
```

Operator redefinition, practice

C++ allows you to redefine many operators in classes. Let's take a look at this using the vector class as an example (DO NOT Confuse with std::vector!), Which stores the value of a 3D vector in space. Let's implement most of the useful operators in it.

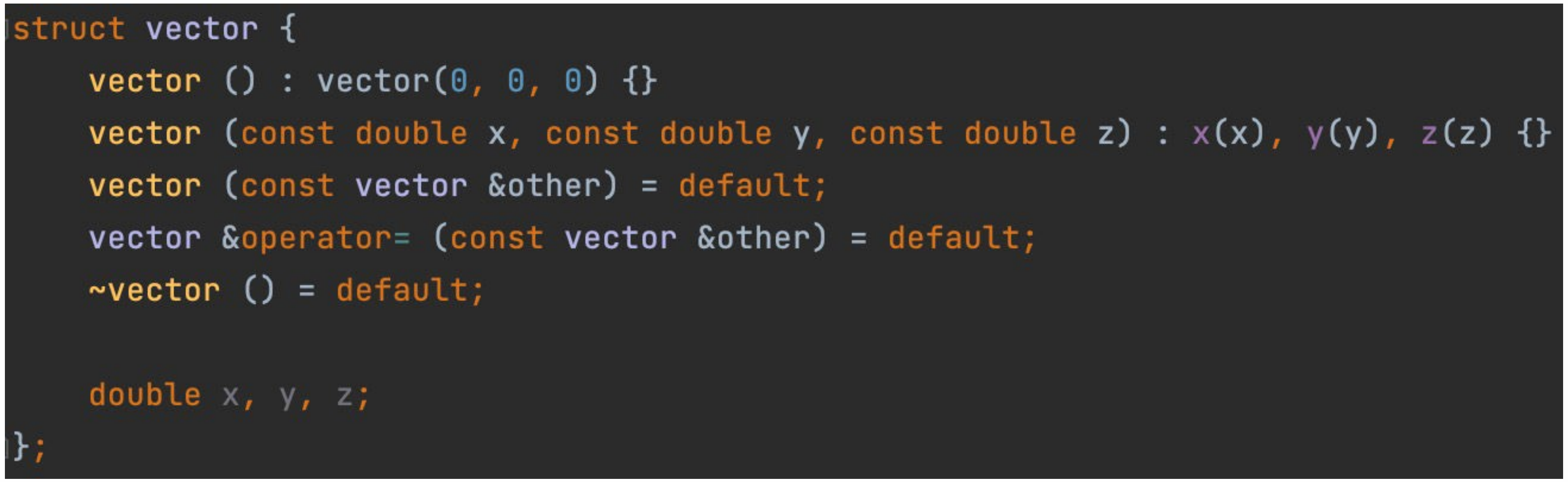


3D Vector: base

First of all, let’s create a class itself. I’ll won’t use encapsulation in order to hide fields x, y and z, because it’s practically useless in this particular case.

Then, I’ll define:

- Constructor from three doubles, which will initialize x, y and z variables with values
- Default constructor, which will initialize coordinates with zeros by calling delegate constructor from three doubles.
- Default copy constructor
- Default copy assignment operator
- Default destructor

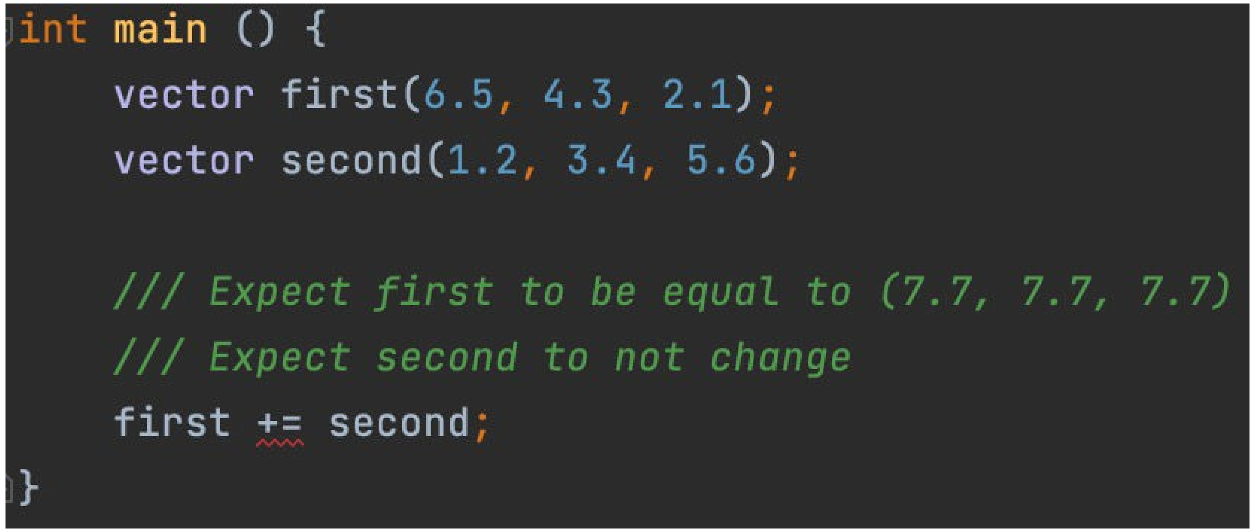


```
struct vector {
    vector () : vector(0, 0, 0) {}
    vector (const double x, const double y, const double z) : x(x), y(y), z(z) {}
    vector (const vector &other) = default;
    vector &operator= (const vector &other) = default;
    ~vector () = default;

    double x, y, z;
};
```

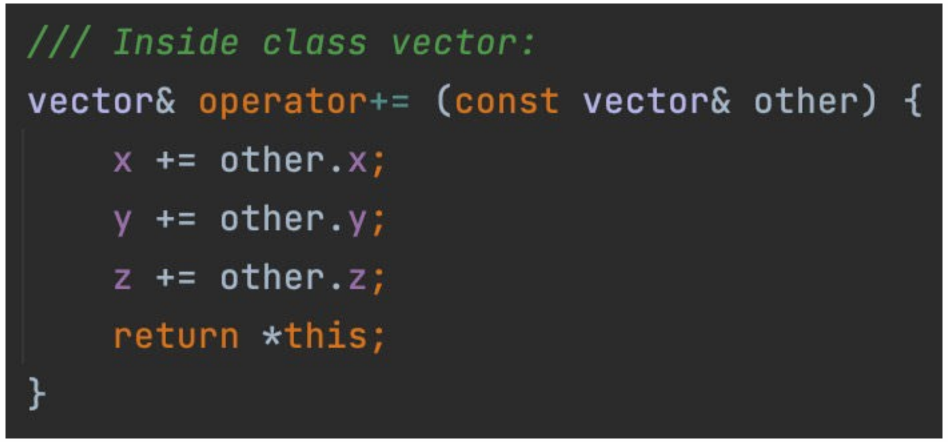
3D Vector: compound assignment operators

First 3D-vector-specific operator, which we'll redefine will be the compound assignment **operator +=**

- Operator += should not modify its second argument (it should receive it by const reference)
- Operator += should be defined in-class and be non-const
- Operator += should return ref to the current object similarly to the copy assignment operator



```cpp
int main () {
    vector first(6.5, 4.3, 2.1);
    vector second(1.2, 3.4, 5.6);

    /// Expect first to be equal to (7.7, 7.7, 7.7)
    /// Expect second to not change
    first += second;
}
```

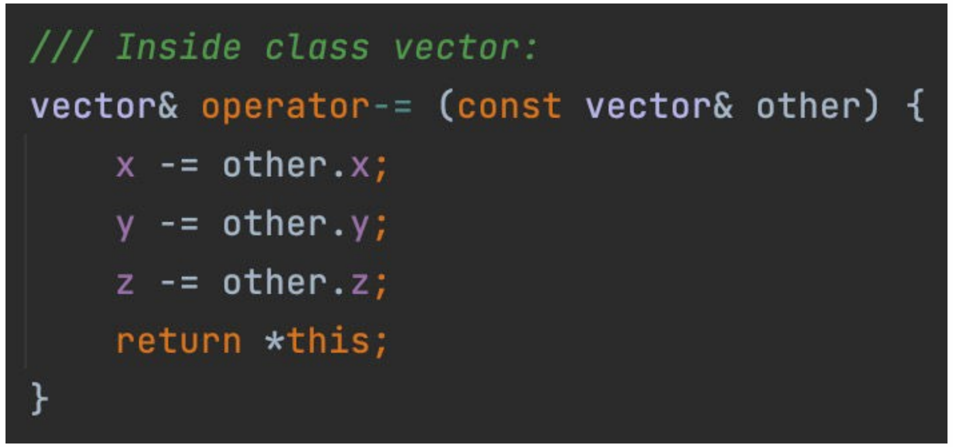


```cpp
/// Inside class vector:
vector& operator+= (const vector& other) {
    x += other.x;
    y += other.y;
    z += other.z;
    return *this;
}
```

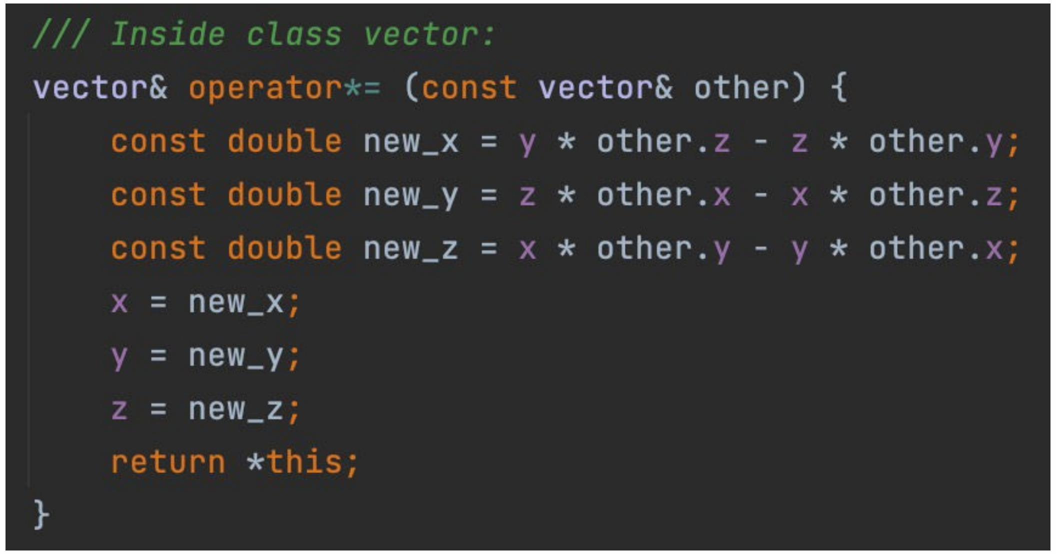
Signature of every the compound assignment operators is similar to copy assignment operator. Implementation is obvious in this case.

3D Vector: compound assignment operators

Implementation of the **operator –=** is very similar.



```
/// Inside class vector:
vector& operator-= (const vector& other) {
    x -= other.x;
    y -= other.y;
    z -= other.z;
    return *this;
}
```

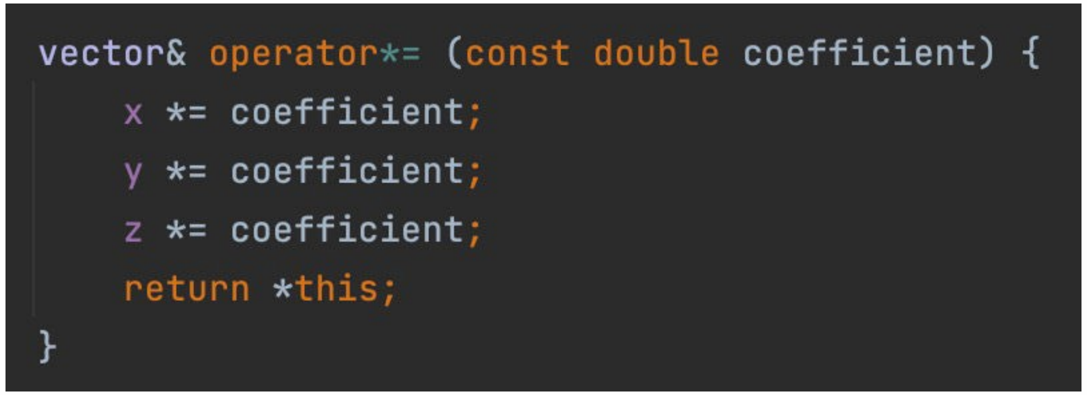


```
/// Inside class vector:
vector& operator*= (const vector& other) {
    const double new_x = y * other.z - z * other.y;
    const double new_y = z * other.x - x * other.z;
    const double new_z = x * other.y - y * other.x;
    x = new_x;
    y = new_y;
    z = new_z;
    return *this;
}
```

However, implementation of the **operator *=** is a little more complicated, just because cross product multiplication is a complicated arithmetic operation.

3D Vector: compound assignment operators

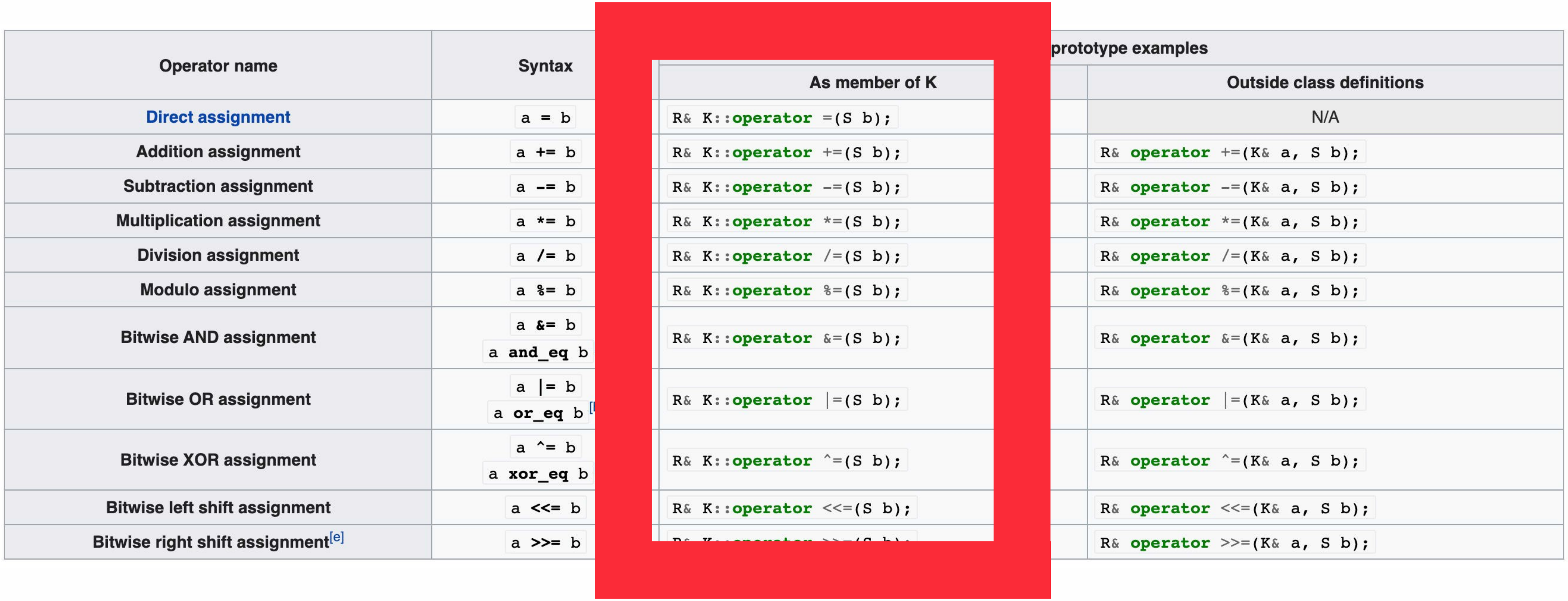
Great, we learned how to multiply a vector by a vector. But what if we want to multiply a vector by a number? Let's define a second method * = that will take a number. One definition rule is not violated in this case, because the signatures are different.



```cpp
vector& operator*= (const double coefficient) {
    x *= coefficient;
    y *= coefficient;
    z *= coefficient;
    return *this;
}
```

3D Vector: compound assignment operators

Summarizing up the **compound assignment operators redefinition** topic, I want to say that the examples I have shown is applicable to other compound assignment operators. Every other compound assignement operator can be redefined simply. Here in the table I gave the signatures of these operators:

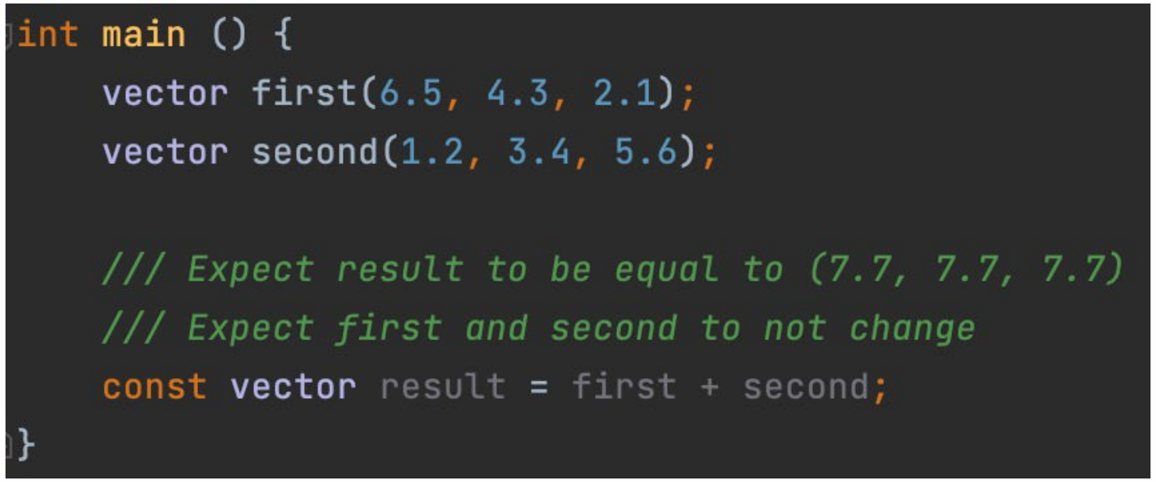


Operator name	Syntax	prototype examples	
		As member of K	Outside class definitions
Direct assignment	`a = b`	`R& K::operator =(S b);`	N/A
Addition assignment	`a += b`	`R& K::operator +=(S b);`	`R& operator +=(K& a, S b);`
Subtraction assignment	`a -= b`	`R& K::operator -=(S b);`	`R& operator -=(K& a, S b);`
Multiplication assignment	`a *= b`	`R& K::operator *=(S b);`	`R& operator *=(K& a, S b);`
Division assignment	`a /= b`	`R& K::operator /=(S b);`	`R& operator /=(K& a, S b);`
Modulo assignment	`a %= b`	`R& K::operator %=(S b);`	`R& operator %=(K& a, S b);`
Bitwise AND assignment	`a &= b` `a and_eq b`	`R& K::operator &=(S b);`	`R& operator &=(K& a, S b);`
Bitwise OR assignment	`a |= b` `a or_eq b`[	`R& K::operator |=(S b);`	`R& operator |=(K& a, S b);`
Bitwise XOR assignment	`a ^= b` `a xor_eq b`	`R& K::operator ^=(S b);`	`R& operator ^=(K& a, S b);`
Bitwise left shift assignment	`a <<= b`	`R& K::operator <<=(S b);`	`R& operator <<=(K& a, S b);`
Bitwise right shift assignment[e]	`a >>= b`	`R& K::operator >>=(S b);`	`R& operator >>=(K& a, S b);`

3D Vector: binary arithmetic operators

Sometimes, we don't want to change the state of the original object when calculating the sum. In this case, we want to create a new object that will be the sum of the two original ones.

There are binary arithmetic operators for this:



```
int main () {
    vector first(6.5, 4.3, 2.1);
    vector second(1.2, 3.4, 5.6);

    /// Expect result to be equal to (7.7, 7.7, 7.7)
    /// Expect first and second to not change
    const vector result = first + second;
}
```

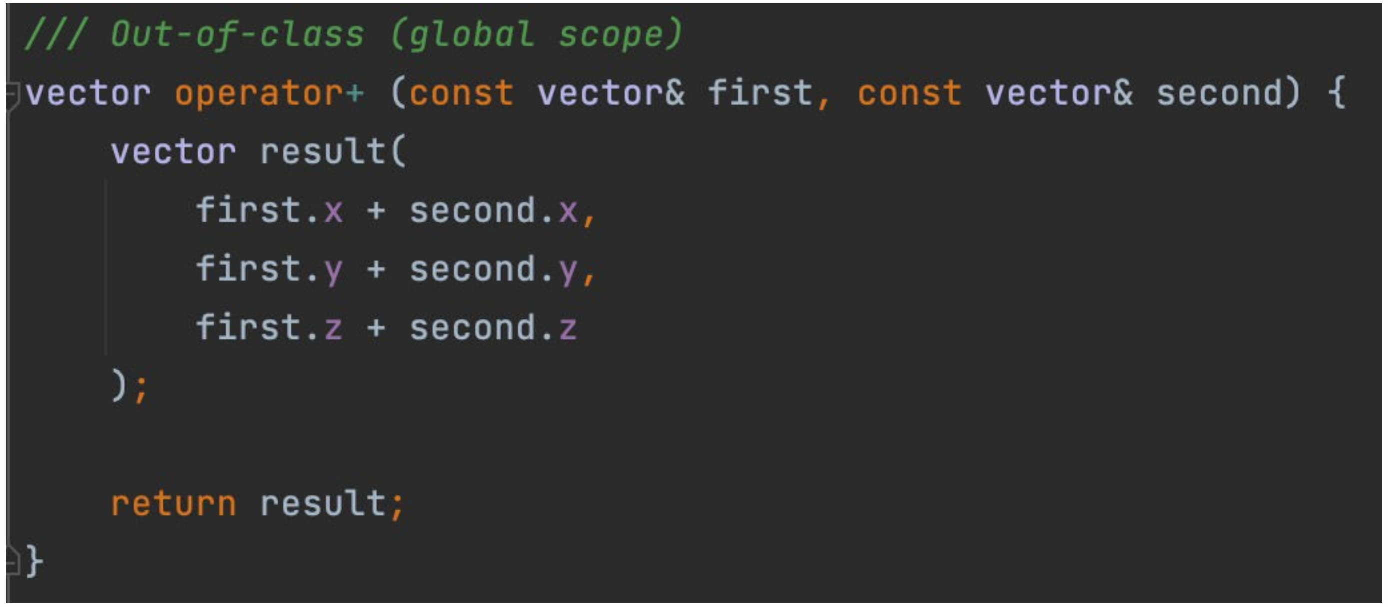
So, binary arithmetic operators should receive two objects by const references and create new object, which stores the sum.

Since the context of the binary operators is not limited to one object, I recommend implementing them out-of-class, this is the way I will show you.

3D Vector: binary arithmetic operators

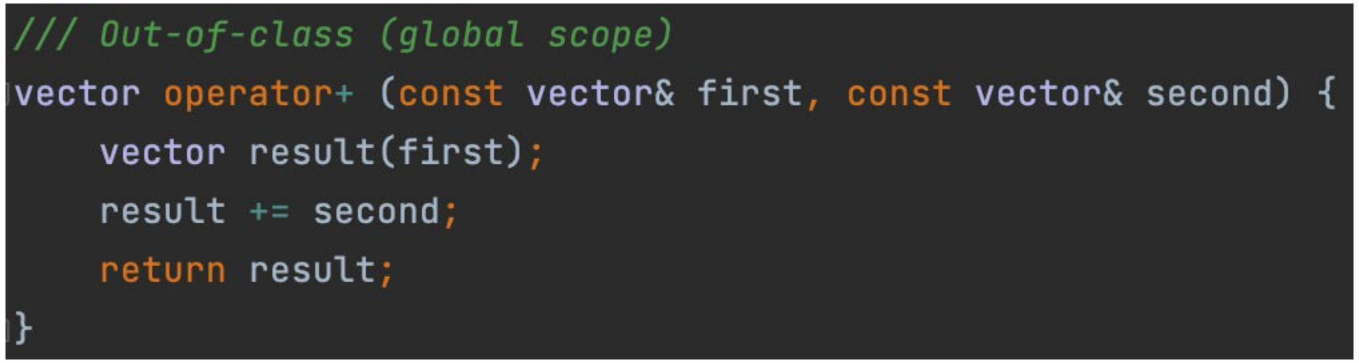
In the screenshot on the right, I have given a common implementation of the binary **operator +**. Everything seems to be fine, however, let's note one *subtlety*: we have defined the logic for summing two vectors **again**! Remember, we have already done this in the case of the **operator +=**! It turns out that in this case we accidently utilized copypasting…

In the vector example, it is not clear why this is bad. But, believe me, in our course in the future we will consider more complex classes that have more sophisticated methods of addition. We would not like to use copypaste in this case. Instead of that, why not rely on the **operator +=**, which we have already implemented?



```
/// Out-of-class (global scope)
vector operator+ (const vector& first, const vector& second) {
    vector result(
        first.x + second.x,
        first.y + second.y,
        first.z + second.z
    );

    return result;
}
```

3D Vector: binary arithmetic operators



```
/// Out-of-class (global scope)
vector operator+ (const vector& first, const vector& second) {
    vector result(first);
    result += second;
    return result;
}
```

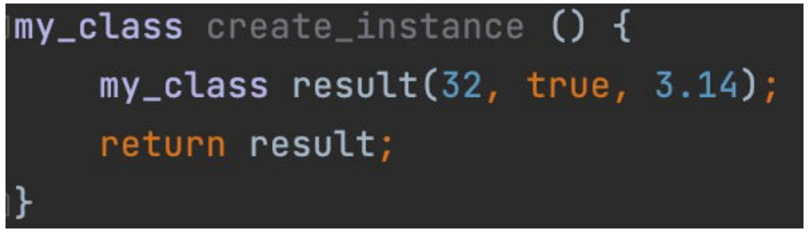
Above is the correctly defined **operator +=** for the vector class. It consists of three parts:

- Making a copy of the first vector
- Calling **operator +=** on this copy
- Returning the resulting value from a function

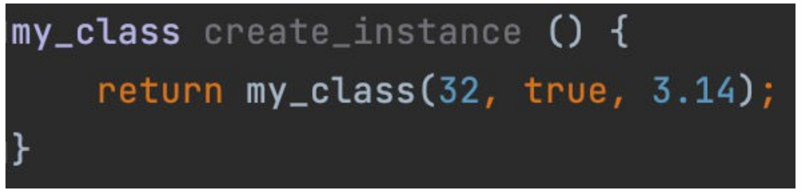
This solution can even be written in one line using the concept of temporary object. Let's look at it in order to optimize our solution.

Temporary object

C ++ allows us to create a temporary unnamed object by calling its constructor:

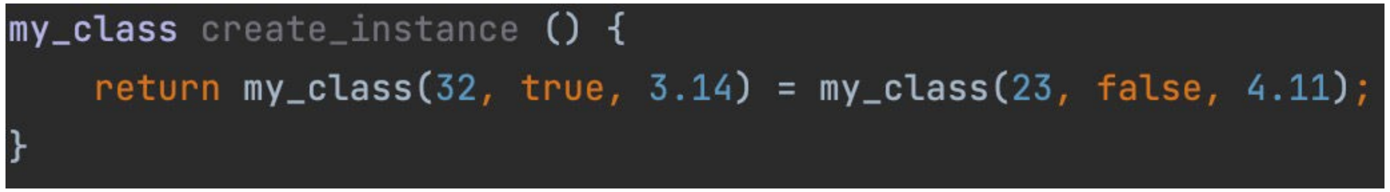


```
my_class create_instance () {
    my_class result(32, true, 3.14);
    return result;
}
```



```
my_class create_instance () {
    return my_class(32, true, 3.14);
}
```

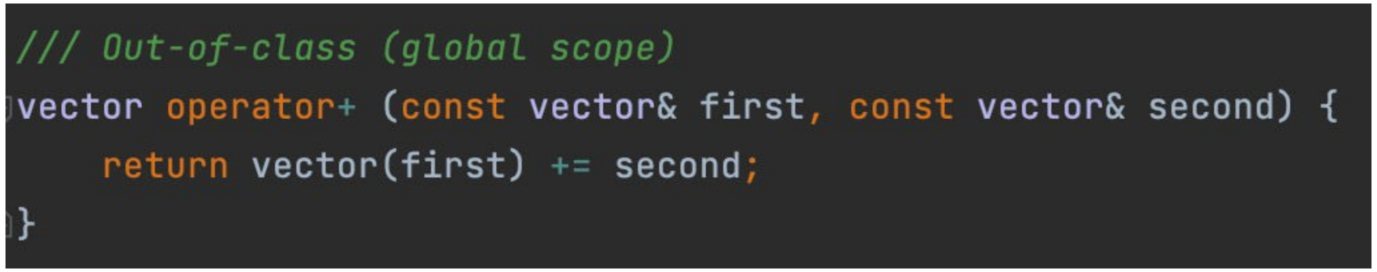
The temporary object can be used both to the left of the assignment operator (lvalue) and to the right (rvalue):



```
my_class create_instance () {
    return my_class(32, true, 3.14) = my_class(23, false, 4.11);
}
```

3D Vector: binary arithmetic operators

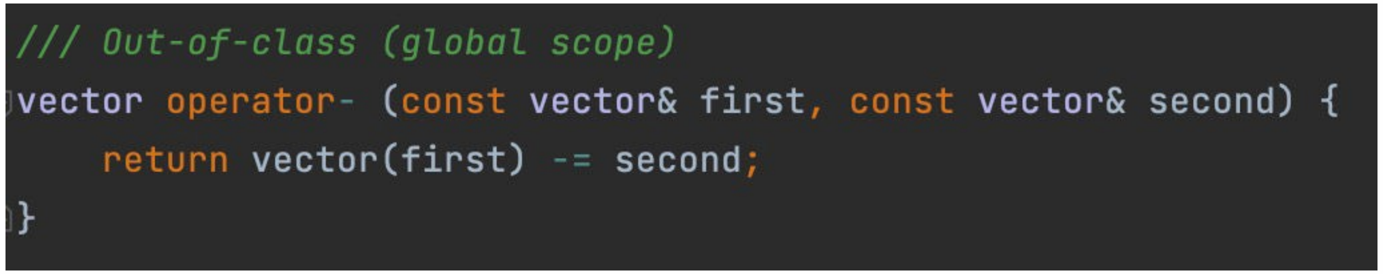
So let's use the concept of a temporary object to implement the binary **operator +**:



```cpp
/// Out-of-class (global scope)
vector operator+ (const vector& first, const vector& second) {
    return vector(first) += second;
}
```

The advantage of this implementation is its versatility. It will almost always be convenient to define the binary **operator +** exactly as shown above.

Operator – and **operator *** can be implemented similarly:



```cpp
/// Out-of-class (global scope)
vector operator- (const vector& first, const vector& second) {
    return vector(first) -= second;
}
```

3D Vector: binary arithmetic operators

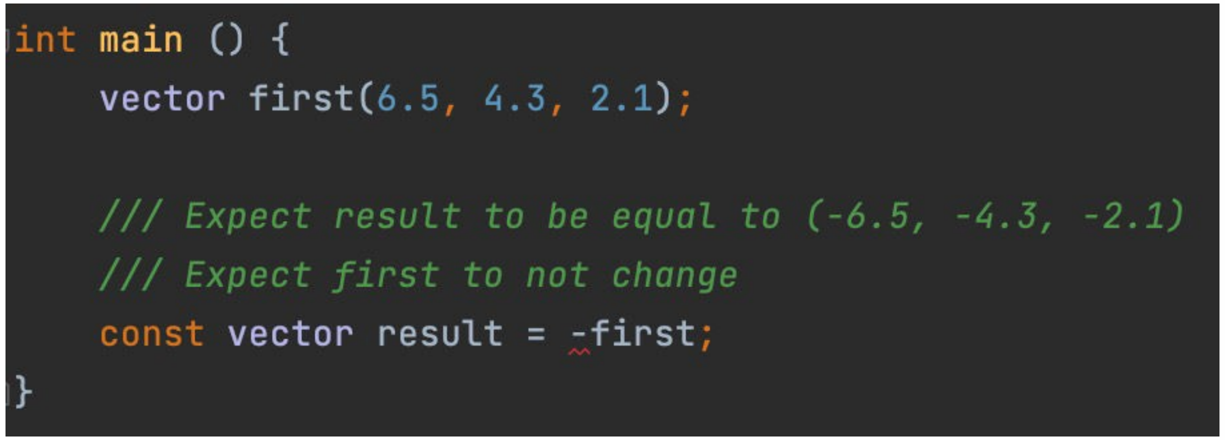
Let’s support the **operator** * in case of:

- multiplication vector by a vector
- multiplication vector by a number
- multiplication number by a vector

```
vector operator* (const vector& first, const vector& second) { return vector(first) *= second; }
vector operator* (const vector& first, const double second) { return vector(first) *= second; }
vector operator* (const double first, const vector& second) { return vector(second) *= first; }
```

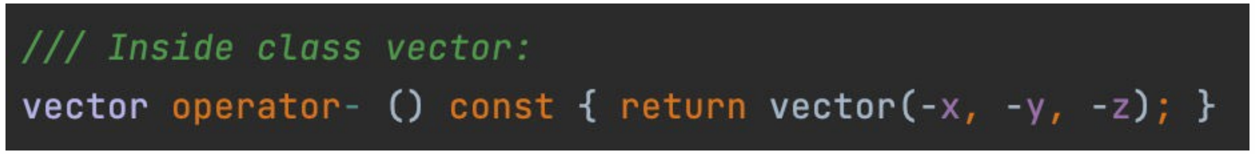
3D Vector: unary arithmetic operators

We may want to get the reflected vector, relative to the origin. In the case of numbers, we use the unary operator – for this task.



```
int main () {
    vector first(6.5, 4.3, 2.1);

    /// Expect result to be equal to (-6.5, -4.3, -2.1)
    /// Expect first to not change
    const vector result = -first;
}
```

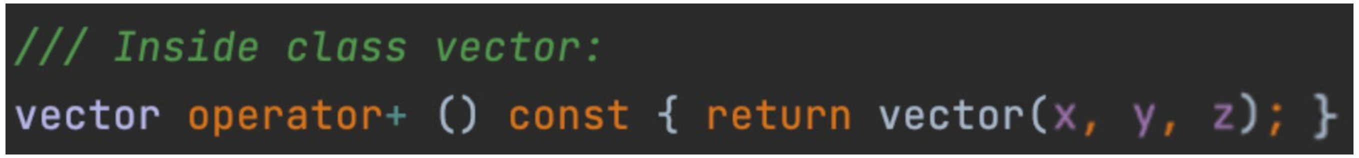
Unary arithmetic operators doesn't change the object they are applied too. Instead, they create a copy of the object. Usually they are defined in-class and, thus, const-qualified:



```
/// Inside class vector:
vector operator- () const { return vector(-x, -y, -z); }
```

3D Vector: unary arithmetic operators

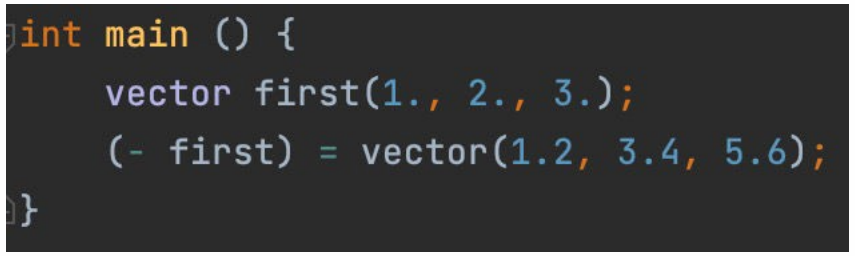
In addition to the unary **operator –**, there is also the unary **operator +**. It might seem like such operator doesn't make sense, because it just copies the value of the vector without changing it. Unfortunately, I cannot yet demonstrate a useful example using this operator, so for now, let's just define it:



```
/// Inside class vector:
vector operator+ () const { return vector(x, y, z); }
```

3D Vector: problem with binary and unary arithmetic operators

Consider the following example:

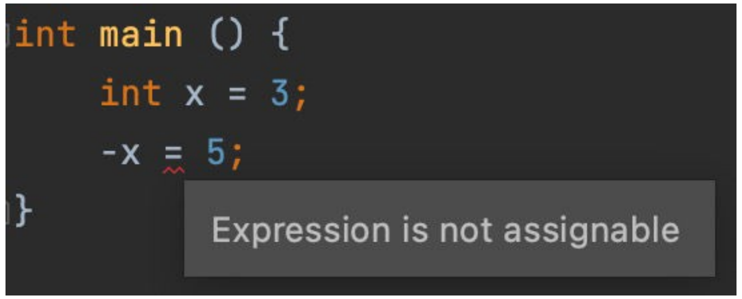


```
int main () {
    vector first(1., 2., 3.);
    (- first) = vector(1.2, 3.4, 5.6);
}
```

On the second line in the main function, we are assigning to a temporary object. Obviously, such an operation is meaningless, because the temporary object will not be available in the future.

This is allowed because the result of a unary **operator –** is an lvalue (a value that can be used as the left operand of an assignment operator) in our case.

By the way, in the case of int, this idea will not work.



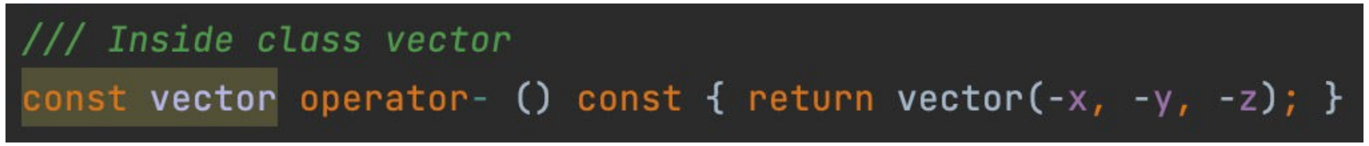
```
int main () {
    int x = 3;
    -x = 5;
}
```

Expression is not assignable

3D Vector: problem with binary and unary arithmetic operators

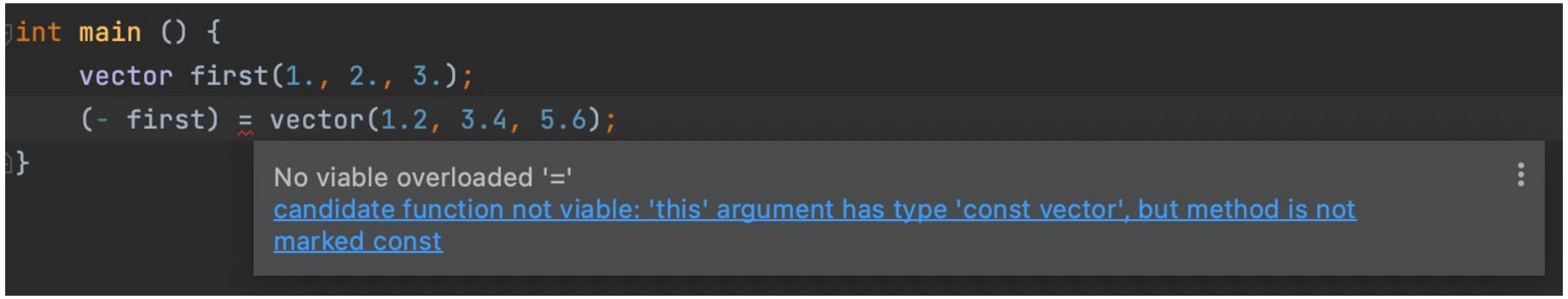
So how do we fix the problem? The obvious solution is to return an rvalue instead of an lvalue from the **operator –**. To do this, just add a const-qualifier to the return type of the **operator –**.

WARNING: do not confuse const-qualifier applied to return type and const-qualifier of a method!



```
/// Inside class vector
const vector operator- () const { return vector(-x, -y, -z); }
```

Well, now the same line of code will not work (which is great) and CE will appear:

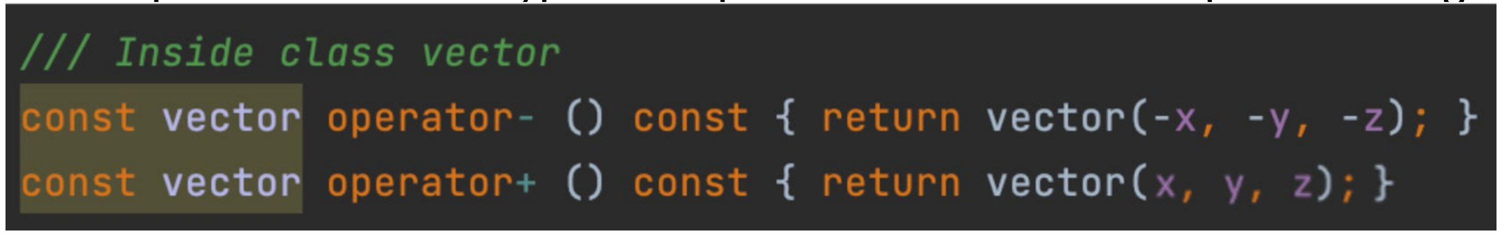


```
int main () {
    vector first(1., 2., 3.);
    (- first) = vector(1.2, 3.4, 5.6);
}
```

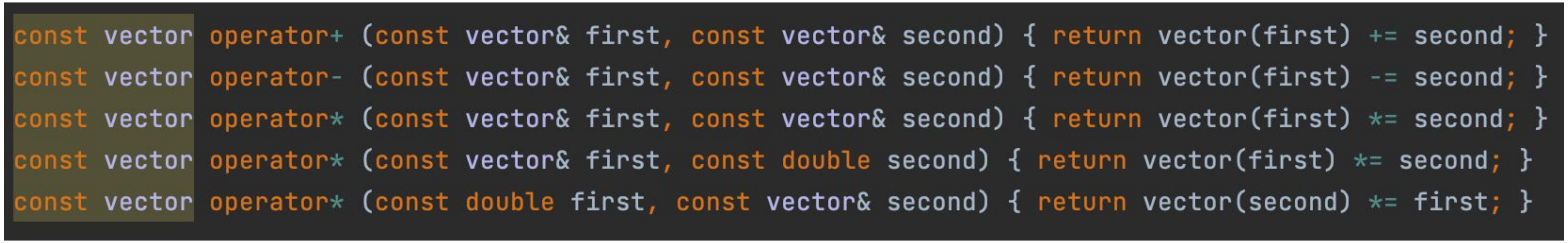
No viable overloaded '='
candidate function not viable: 'this' argument has type 'const vector', but method is not marked const

3D Vector: problem with binary and unary arithmetic operators

Let's update the return type in all places where a similar problem might arise.



```
/// Inside class vector
const vector operator- () const { return vector(-x, -y, -z); }
const vector operator+ () const { return vector(x, y, z); }
```

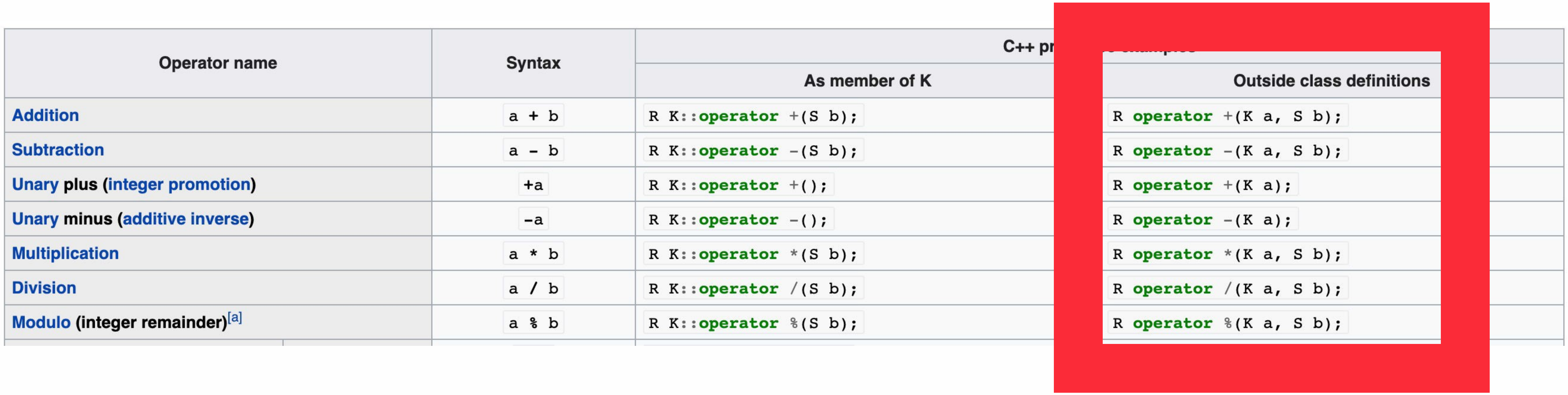


```
const vector operator+ (const vector& first, const vector& second) { return vector(first) += second; }
const vector operator- (const vector& first, const vector& second) { return vector(first) -= second; }
const vector operator* (const vector& first, const vector& second) { return vector(first) *= second; }
const vector operator* (const vector& first, const double second) { return vector(first) *= second; }
const vector operator* (const double first, const vector& second) { return vector(second) *= first; }
```

Note: CLion marked the return type in yellow because this solution is not entirely correct. We'll dive into details later when we will be discussing the so-called reference qualifiers. Later, we will come up with a correct solution of the problem using them. The current solution should be taken as temporary.

3D Vector: arithmetic operators

Summarizing up the **arithmetic operators redefinition** topic, I want to show you the table below:



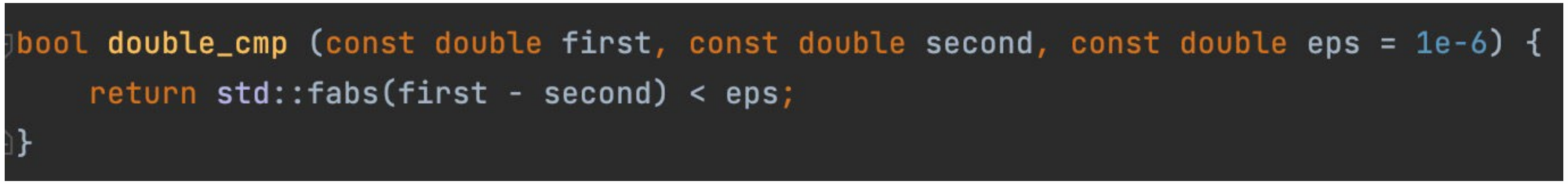
Operator name	Syntax	C++ pr	
		As member of K	Outside class definitions
Addition	`a + b`	`R K::operator +(S b);`	`R operator +(K a, S b);`
Subtraction	`a - b`	`R K::operator -(S b);`	`R operator -(K a, S b);`
Unary plus (integer promotion)	`+a`	`R K::operator +();`	`R operator +(K a);`
Unary minus (additive inverse)	`-a`	`R K::operator -();`	`R operator -(K a);`
Multiplication	`a * b`	`R K::operator *(S b);`	`R operator *(K a, S b);`
Division	`a / b`	`R K::operator /(S b);`	`R operator /(K a, S b);`
Modulo (integer remainder)[a]	`a % b`	`R K::operator %(S b);`	`R operator %(K a, S b);`

Operators **operator /** and **operator %** can be redefined similarly to operators **operator +**, **operator –** and **operator ***.

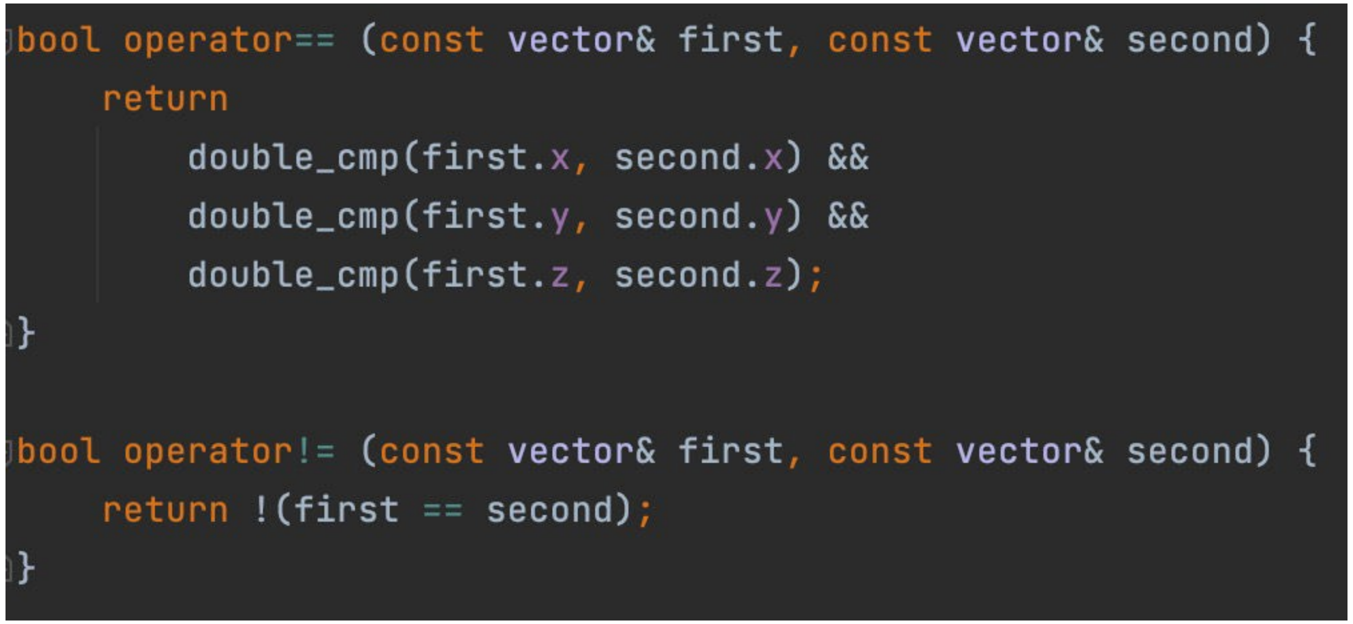
Just don’t forget that every binary arithmetic operator should be implemented via a corresponding compound assignment operator.

3D Vector: comparison operators

Let's define two helper operators that allow us to compare two vectors with each other: the **operator ==** and the **operator !=**. Since the coordinates of the vector are real numbers, we will compare them using the algorithm for comparing real numbers.



```
bool double_cmp (const double first, const double second, const double eps = 1e-6) {
    return std::fabs(first - second) < eps;
}
```



```
bool operator== (const vector& first, const vector& second) {
    return
        double_cmp(first.x, second.x) &&
        double_cmp(first.y, second.y) &&
        double_cmp(first.z, second.z);
}

bool operator!= (const vector& first, const vector& second) {
    return !(first == second);
}
```

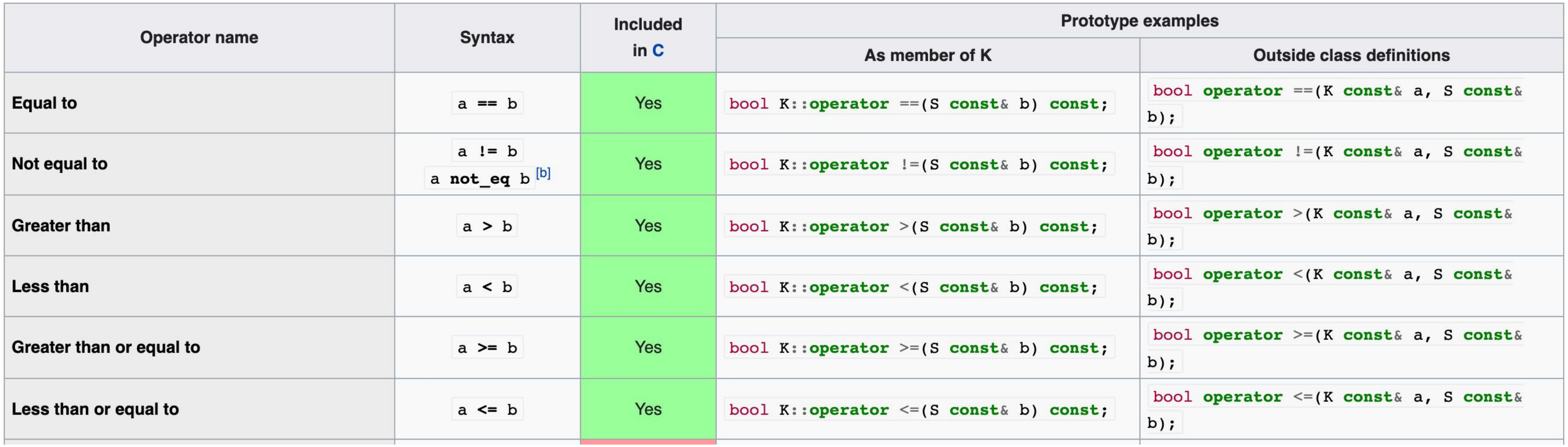
Please pay attention to the implementation of the **operator !=**.

It is very important that it refers to the **operator ==**. Thus, we avoid the tautology (just like with the example about **operator +=** and **operator +**)

3D Vector: comparison operators

The remaining four comparison operators can also be redefined, however there is a trick that we will discuss later when we'll redefine these operators.

Unfortunately, in the case of vector, we can't define these four operators, because the ordering relation over points in the $\mathbf{R}^{!}$ space does not exist.

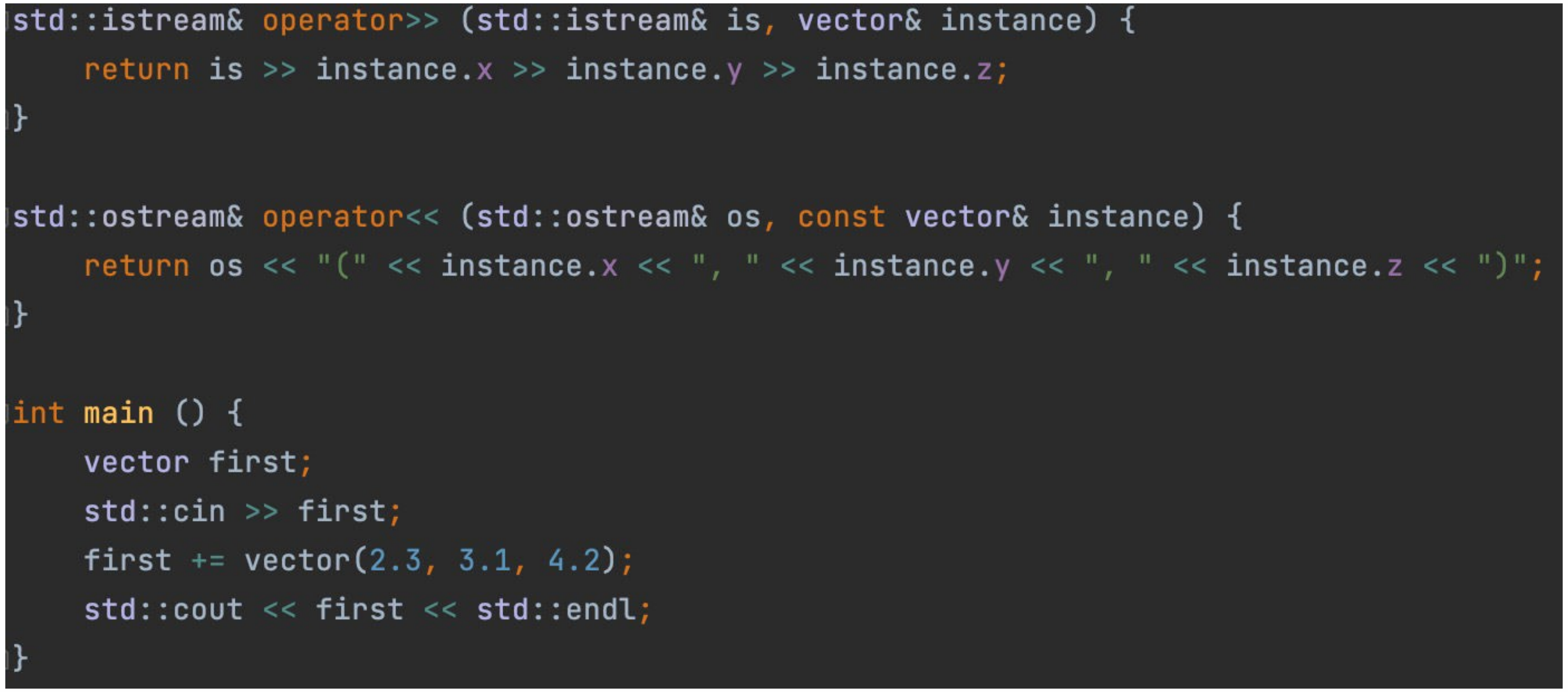


Operator name	Syntax	Included in C	Prototype examples	
			As member of K	Outside class definitions
Equal to	`a == b`	Yes	`bool K::operator ==(S const& b) const;`	`bool operator ==(K const& a, S const& b);`
Not equal to	`a != b` `a not_eq b` [b]	Yes	`bool K::operator !=(S const& b) const;`	`bool operator !=(K const& a, S const& b);`
Greater than	`a > b`	Yes	`bool K::operator >(S const& b) const;`	`bool operator >(K const& a, S const& b);`
Less than	`a < b`	Yes	`bool K::operator <(S const& b) const;`	`bool operator <(K const& a, S const& b);`
Greater than or equal to	`a >= b`	Yes	`bool K::operator >=(S const& b) const;`	`bool operator >=(K const& a, S const& b);`
Less than or equal to	`a <= b`	Yes	`bool K::operator <=(S const& b) const;`	`bool operator <=(K const& a, S const& b);`

3D Vector: I/O operators

If we want our object to be scannable from the terminal and printable into it, then we have to redefine two binary bitwise shift operators of a special form (accepting **std::istream** and **std::ostream**, representing input and output streams, respectively).

Both of these operators must return the reference to the stream they received as an argument. This is necessary in order to support the chain call of **operator >>** and **operator <<**



```cpp
std::istream& operator>> (std::istream& is, vector& instance) {
    return is >> instance.x >> instance.y >> instance.z;
}

std::ostream& operator<< (std::ostream& os, const vector& instance) {
    return os << "(" << instance.x << ", " << instance.y << ", " << instance.z << ")";
}

int main () {
    vector first;
    std::cin >> first;
    first += vector(2.3, 3.1, 4.2);
    std::cout << first << std::endl;
}
```