21.9 — Overloading the subscript operator

ALEX DECEMBER 7, 2023

When working with arrays, we typically use the subscript operator ([]) to index specific elements of an array:

```
myArray[0] = 7; // put the value 7 in the first element of the array
```

However, consider the following `IntList` class, which has a member variable that is an array:

```
class IntList
{
private:
    int m_list[10]{};
};

int main()
{
    IntList list{};
    // how do we access elements from m_list?
    return 0;
}
```

Because the m_list member variable is private, we can not access it directly from variable list. This means we have no way to directly get or set values in the m_list array. So how do we get or put elements into our list?

Without operator overloading, the typical method would be to create access functions:

```
class IntList
{
private:
    int m_list[10]{};

public:
    void setItem(int index, int value) { m_list[index] = value; }
    int getItem(int index) const { return m_list[index]; }
};
```

While this works, it's not particularly user friendly. Consider the following example:

```
int main()
{
    IntList list{};
    list.setItem(2, 3);

    return 0;
}
```

Are we setting element 2 to the value 3, or element 3 to the value 2? Without seeing the definition of `setItem()`, it's simply not clear.

You could also just return the entire list and use operator[] to access the element:

```
class IntList
{
private:
    int m_list[10]{};

public:
    int* getList() { return m_list; }
};
```

While this also works, it's syntactically odd:

```
int main()
{
    IntList list{};
    list.getList()[2] = 3;

    return 0;
}
```

Overloading operator[]

However, a better solution in this case is to overload the subscript operator ([]) to allow access to the elements of m_list. The subscript operator is one of the operators that must be overloaded as a member function. An overloaded operator[] function will always take one parameter: the subscript that the user places between the hard braces. In our IntList case, we expect the user to pass in an integer index, and we'll return an integer value back as a result.

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index)
    {
        return m_list[index];
    }
};

/*
// Can also be implemented outside the class definition
int& IntList::operator[] (int index)
{
    return m_list[index];
}
*/

int main()
{
    IntList list{};
    list[2] = 3; // set a value
    std::cout << list[2] << '\n'; // get a value

    return 0;
}
```

Now, whenever we use the subscript operator ([]) on an object of our class, the compiler will return the corresponding element from the m_list member variable! This allows us to both get and set values of m_list directly.

This is both easy syntactically and from a comprehension standpoint. When `list[2]` evaluates, the compiler first checks to see if there's an overloaded operator[] function. If so, it passes the value inside the hard braces (in this case, 2) as an argument to the function.

Note that although you can provide a default value for the function parameter, actually using operator[] without a subscript inside is not considered a valid syntax, so there's no point.

Tip

C++23 adds support for overloading operator[] with multiple subscripts.

Why operator[] returns a reference

Let's take a closer look at how `list[2] = 3` evaluates. Because the subscript operator has a higher precedence than the assignment operator, `list[2]` evaluates first. `list[2]` calls operator[], which we've defined to return a reference to `list.m_list[2]`. Because operator[] is returning a reference, it returns the actual `list.m_list[2]` array element. Our partially evaluated expression becomes `list.m_list[2] = 3`, which is a straightforward integer assignment.

In lesson 12.2 -- Value categories (lvalues and rvalues) (https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/), you learned that any value on the left hand side of an assignment statement must be an l-value (which is a variable that has an actual memory address). Because the result of operator[] can be used on the left hand side of an assignment (e.g. `list[2] = 3`), the return value of operator[] must be an l-value. As it turns out, references are always l-values, because you can only take a reference of variables that have memory addresses. So by returning a reference, the compiler is satisfied that we are returning an l-value.

Consider what would happen if operator[] returned an integer by value instead of by reference. `list[2]` would call operator[], which would return the value of list.m_list[2]. For example, if m_list[2] had the value of 6, operator[] would return the value 6. `list[2] = 3` would partially evaluate to `6 = 3`, which makes no sense! If you try to do this, the C++ compiler will complain:

```
C:VCProjectsTest.cpp(386) : error C2106: '=' : left operand must be l-value
```

Overloaded operator[] for const objects

In the above IntList example, operator[] is non-const, and we can use it as an l-value to change the state of non-const objects. However, what if our IntList object was const? In this case, we wouldn't be able to call the non-const version of operator[] because that would allow us to potentially change the state of a const object.

The good news is that we can define a non-const and a const version of operator[] separately. The non-const version will be used with non-const objects, and the const version with const-objects.

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for this example

public:
    // For non-const objects: can be used for assignment
    int& operator[] (int index)
    {
        return m_list[index];
    }

    // For const objects: can only be used for access
    // This function could also return by value if the type is cheap to copy
    const int& operator[] (int index) const;
    {
        return m_list[index];
    }
};

int main()
{
    IntList list{};
    list[2] = 3; // okay: calls non-const version of operator[]
    std::cout << list[2] << '\n';

    const IntList clist{};
    // clist[2] = 3; // compile error: clist[2] returns const refrence, which we can't assign to
    std::cout << clist[2] << '\n';

    return 0;
}
```

Removing duplicate code between const and non-const overloads

In the above example, note that the implementations of `int& IntList::operator[](int)` and `const int& IntList::operator[](int) const` are identical. The only difference is the return type of the function.

In cases where the implementation is trivial (e.g. a single line), it's fine (and preferred) to have both functions use an identical implementation. The small amount of redundancy this introduces isn't worth removing.

But what if the implementation of these operators was complex, requiring many statements? For example, maybe it's important that we validate that the index is actually valid, which requires adding many redundant lines of code to each function.

In such a case, the redundancy introduced by having many duplicate statements is more problematic, and it would be desirable to have a single implementation that we could use for both overloads. But how? Normally we'd simply implement one function in terms of the other (e.g. have one function call the other). But that's a bit tricky in this case. The const version of the function can't call the non-const version of the function, because that would require discarding the const of a const object. And while the non-const version of the function can call the const version of the function, the const version of the function returns a const reference, when we need to return a non-const reference. Fortunately, there is a way to work around this.

The preferred solution is as follows:

- Refactor the logic into another function (usually a private const member function, but could be a non-member function).
- Have the non-const function call the const function and use `const_cast` to remove the const of the returned reference.

The resulting solution looks something like this:

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for this example

    // Use private const member function to handle duplicate logic
    const int& getIndex(int index) const
    {
        // Complex code goes here
        return m_list[index];
    }

public:
    // These overloaded operators can now be implemented as a single line,
    // helping to highlight the actual difference between them
    int& operator[] (int index)
    {
        // Since we know our implicit object is non-const
        // We can strip the const off the reference returned by getIndex
        return const_cast<int&>(getIndex(index));
    }

    const int& operator[] (int index) const
    {
        return getIndex(index);
    }
};

int main()
{
    IntList list{};
    list[2] = 3; // okay: calls non-const version of operator[]
    std::cout << list[2] << '\n';

    const IntList clist{};
    // clist[2] = 3; // compile error: clist[2] returns const refrence, which we can't assign to
    std::cout << clist[2] << '\n';

    return 0;
}
```

Normally using `const_cast` to remove const is something we want to avoid, but in this case it's acceptable. If the non-const overload was called, then we know we're working with a non-const object. It's okay to remove the const on a const reference to a non-const object.

For advanced readers

In C++23, we can do even better by making use of several features we don't yet cover in this tutorial series:

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for this example

public:
    // Use an explicit object parameter (self) and auto&& to differentiate const vs non-const
    auto&& operator[](this auto&& self, int index)
    {
        // Complex code goes here
        return self.m_list[index];
    }
};

int main()
{
    IntList list{};
    list[2] = 3; // okay: calls non-const version of operator[]
    std::cout << list[2] << '\n';

    const IntList clist{};
    // clist[2] = 3; // compile error: clist[2] returns const refrence, which we can't assign to
    std::cout << clist[2] << '\n';

    return 0;
}
```

Detecting index validity

One other advantage of overloading the subscript operator is that we can make it safer than accessing arrays directly. Normally, when accessing arrays, the subscript operator does not check whether the index is valid. For example, the compiler will not complain about the following code:

```
int list[5]{};
list[7] = 3; // index 7 is out of bounds!
```

However, if we know the size of our array, we can make our overloaded subscript operator check to ensure the index is within bounds:

```
#include <cassert> // for assert()
#include <iterator> // for std::size()

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index)
    {
        assert(index >= 0 && static_cast<std::size_t>(index) < std::size(m_list));

        return m_list[index];
    }
};
```

In the above example, we have used the assert() function (included in the cassert header) to make sure our index is valid. If the expression inside the assert evaluates to false (which means the user passed in an invalid index), the program will terminate with an error message, which is much better than the alternative (corrupting memory). This is probably the most common method of doing error checking of this sort.

If you don't want to use an assert (which will be compiled out of a non-debug build) you can instead use an if-statement and your favorite error

handling method (e.g. throw an exception, call `std::exit`, etc...):

```
#include <iterator> // for std::size()

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index)
    {
        if (!(index >= 0 && static_cast<std::size_t>(index) < std::size(m_list))
        {
            // handle invalid index here
        }

        return m_list[index];
    }
};
```

Pointers to objects and overloaded operator[] don't mix

If you try to call operator[] on a pointer to an object, C++ will assume you're trying to index an array of objects of that type.

Consider the following example:

```
#include <cassert> // for assert()
#include <iterator> // for std::size()

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index)
    {
        return m_list[index];
    }
};

int main()
{
    IntList* list{ new IntList{} };
    list [2] = 3; // error: this will assume we're accessing index 2 of an array of IntLists
    delete list;

    return 0;
}
```

Because we can't assign an integer to an IntList, this won't compile. However, if assigning an integer was valid, this would compile and run, with undefined results.

> **Rule**
>
> Make sure you're not trying to call an overloaded operator[] on a pointer to an object.

The proper syntax would be to dereference the pointer first (making sure to use parenthesis since operator[] has higher precedence than operator*), then call operator[]:

```
int main()
{
    IntList* list{ new IntList{} };
    (*list)[2] = 3; // get our IntList object, then call overloaded operator[]
    delete list;

    return 0;
}
```

This is ugly and error prone. Better yet, don't set pointers to your objects if you don't have to.

The function parameter does not need to be an integral type

As mentioned above, C++ passes what the user types between the hard braces as an argument to the overloaded function. In most cases, this will be an integral value. However, this is not required -- and in fact, you can define that your overloaded operator[] take a value of any type you desire. You could define your overloaded operator[] to take a double, a std::string, or whatever else you like.

As a ridiculous example, just so you can see that it works:

```
#include <iostream>
#include <string_view> // C++17

class Stupid
{
private:

public:
    void operator[] (std::string_view index);
};

// It doesn't make sense to overload operator[] to print something
// but it is the easiest way to show that the function parameter can be a non-integer
void Stupid::operator[] (std::string_view index)
{
    std::cout << index;
}

int main()
{
    Stupid stupid{};
    stupid["Hello, world!"];

    return 0;
}
```

As you would expect, this prints:

```
Hello, world!
```

Overloading operator[] to take a std::string parameter can be useful when writing certain kinds of classes, such as those that use words as indices.

Quiz time

Question #1

A map is a class that stores elements as a key-value pair. The key must be unique, and is used to access the associated pair. In this quiz, we're going to write an application that lets us assign grades to students by name, using a simple map class. The student's name will be the key, and the grade (as a char) will be the value.

a) First, write a struct named `StudentGrade` that contains the student's name (as a `std::string`) and grade (as a `char`).

Show Solution (javascript:void(0))

b) Add a class named `GradeMap` that contains a `std::vector` of `StudentGrade` named `m_map`.

Show Solution (javascript:void(0))

c) Write an overloaded `operator[]` for this class. This function should take a `std::string` parameter, and return a reference to a char. In the body of the function, first see if the student's name already exists (You can use `std::find_if` from <algorithm>). If the student exists, return a reference to the grade and you're done. Otherwise, use the `std::vector::push_back()` function to add a `StudentGrade` for this new student. When you do this, `std::vector` will add a copy of your `StudentGrade` to itself (resizing if needed, invalidating all previously returned references). Finally, we need to return a reference to the grade for the student we just added to the `std::vector`. We can access the student we just added using the `std::vector::back()` function.

The following program should run:

```
#include <iostream>

// ...

int main()
{
    GradeMap grades{};

    grades["Joe"] = 'A';
    grades["Frank"] = 'B';

    std::cout << "Joe has a grade of " << grades["Joe"] << '\n';
    std::cout << "Frank has a grade of " << grades["Frank"] << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

Question #2

Extra credit #1: The `GradeMap` class and sample program we wrote is inefficient for many reasons. Describe one way that the `GradeMap` class could be improved.

Show Solution (javascript:void(0))

Question #3

Extra credit #2: Why does this program potentially not work as expected?

```
#include <iostream>

int main()
{
    GradeMap grades{};

    char& gradeJoe{ grades["Joe"] }; // does a push_back
    gradeJoe = 'A';

    char& gradeFrank{ grades["Frank"] }; // does a push_back
    gradeFrank = 'B';

    std::cout << "Joe has a grade of " << gradeJoe << '\n';
    std::cout << "Frank has a grade of " << gradeFrank << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

Next lesson
21.10 Overloading the parenthesis operator

Back to table of contents

Previous lesson
21.8 Overloading the increment and decrement operators

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/ are connected to your provided email address.

411 COMMENTS

Newest