9.8 — Overloading the subscript operator

BY ALEX ON OCTOBER 19TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 24TH, 2018

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) { 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.

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

public:
    int& operator[] (const int index);
};

int& IntList::operator[] (const int index)
{
    return m_list[index];
}
```

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:

```
    IntList list;
    list[2] = 3; // set a value
    std::cout << list[2]; // get a value

    return 0;
```

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.

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 the lesson **a first look at variables**, 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
```

Dealing with 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.

```
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 e
xample

public:
    int& operator[] (const int index);
    const int& operator[] (const int index) const;
};

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

const int& IntList::operator[] (const int index) const // for const objects: can only be used for acces
s
{
    return m_list[index];
}

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

    const IntList clist;
    clist[2] = 3; // compile error: calls const version of operator[], which returns a const referenc
e.  Cannot assign to this.
    std::cout << clist[2];

    return 0;
}
```

If we comment out the line `clist[2] = 3`, the above program compiles and executes as expected.

Error checking

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()

class IntList
{
private:
    int m_list[10];

public:
    int& operator[] (const int index);
};

```

```
int& IntList::operator[] (const int index)
{
    assert(index >= 0 && index < 10);

    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.

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()

class IntList
{
private:
    int m_list[10];

public:
    int& operator[] (const int index);
};

int& IntList::operator[] (const int index)
{
    assert(index >= 0 && index < 10);

    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 integer

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 integer 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>

class Stupid
{
private:

public:
    void operator[] (std::string 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 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.

Conclusion

The subscript operator is typically overloaded to provide direct access to individual elements from an array (or other similar structure) contained within a class. Because strings are often implemented as arrays of characters, operator[] is often implemented in string classes to allow the user to access a single character of the string.

Quiz time

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.

1a) First, write a struct named StudentGrade that contains the student's name (as a std::string) and grade (as a char).

Hide Solution

```
#include <string>
struct StudentGrade
{
    std::string name;
    char grade;
};
```

1b) Add a class named GradeMap that contains a std::vector of StudentGrade named m_map. Add a default constructor that does nothing.

Hide Solution

```
#include <string>
#include <vector>

struct StudentGrade
{
    std::string name;
    char grade;
};

class GradeMap
{
private:
    std::vector<StudentGrade> m_map;

public:
    GradeMap()
    {
    }
};
```

1c) 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 iterate through the vector to see if the student's name already exists (you can use a for-each loop for this). 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). 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;
}
```

Hide Solution

```
#include <iostream>
#include <string>
#include <vector>

struct StudentGrade
{
    std::string name;
    char grade;
};

class GradeMap
{
private:
    std::vector<StudentGrade> m_map;

public:
    GradeMap()
    {
    }

    char& operator[](const std::string &name);
```

```
};

char& GradeMap::operator[](const std::string &name)
{
    // See if we can find the name in the vector
    for (auto &ref : m_map)
    {
        // If we found the name in the vector, return the grade
        if (ref.name == name)
            return ref.grade;
    }

    // otherwise create a new StudentGrade for this student
    StudentGrade temp { name, ' ' };

    // otherwise add this to the end of our vector
    m_map.push_back(temp);

    // and return the element
    return m_map.back().grade;
}

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;
}
```

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.

Hide Solution

std::vector is unsorted by nature. This means every time we call operator[], we're potentially traversing the entire std::vector to find our element. With a few elements, this isn't a problem, but as we continue to add names, this will become increasingly slow. We could optimize this by keeping our m_map sorted and using a binary search, so we minimize the number of elements we have to look through to find the ones we're interested in.

3) Extra credit #2: Why doesn't this program 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;
}
```

Hide Solution

When Frank is added, the std::vector must grow to hold it. This requires dynamically allocating a new block of memory, copying the elements in the array to that new block, and deleting the old block. When this happens, any references to existing elements in the

std::vector are invalidated! In other words, after we push_back("Frank"), gradeJoe is left as a dangling reference to deleted memory. This will lead to undefined results.


9.9 -- Overloading the parenthesis operator


Index


9.7 -- Overloading the increment and decrement operators

Share this:

C++ TUTORIAL | PRINT THIS POST

143 comments to 9.8 — Overloading the subscript operator

« Older Comments 1 2

Vamshi malreddy
July 11, 2018 at 5:06 pm · Reply

what's wrong with this for overloading operator[] in Q 1.C of the quiz ??

```
char& operator[](string str)
    {
        for(StudentGrade &student: m_map)
        {
            if(student.m_name == str )
                return student.m_grade;
        }

        m_map.push_back(StudentGrade(str));
        return (*this)[str];
    }
```