9.8 — Overloading the subscript operator

BY ALEX ON OCTOBER 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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.

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

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

int& IntList::operator[] (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] << '\n'; // 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 lesson **1.3 -- Introduction to 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.

```
1  class IntList
2  {
3  private:
4      int m_list[]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for th
5
6  public:
7      int& operator[] (int index);
8      const int& operator[] (int index) const;
9  };
10
11 int& IntList::operator[] (int index) // for non-const objects: can be used for assignment
12 {
13     return m_list[index];
14 }
15
16 const int& IntList::operator[] (int index) const // for const objects: can only be used for ac
17 {
18     return m_list[index];
19 }
20
21 int main()
22 {
23     IntList list{};
24     list[2] = 3; // okay: calls non-const version of operator[]
25     std::cout << list[2] << '\n';
26
27     const IntList clist{};
28     clist[2] = 3; // compile error: calls const version of operator[], which returns a const r
29     std::cout << clist[2] << '\n';
30
31     return 0;
32 }
```

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[] (int index);
};

int& IntList::operator[] (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[] (int index);
};

int& IntList::operator[] (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[] (const 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[] (const 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

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

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

Show Solution

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';
```

```
11
12      std::cout << "Joe has a grade of " << grades["Joe"] << '\n';
13      std::cout << "Frank has a grade of " << grades["Frank"] << '\n';
14
15      return 0;
16  }
```

Show Solution

Tip

Since maps are common, the standard library offers **`std::map`**, which is not currently covered on learncpp. Using `std::map`, we can simplify our code to

```
1   #include <iostream>
2   #include <map> // std::map
3   #include <string>
4
5   int main()
6   {
7       // std::map can be initialized
8       std::map<std::string, char> grades{
9           { "Joe", 'A' },
10          { "Frank", 'B' }
11      };
12
13      // and assigned
14      grades["Susan"] = 'C';
15      grades["Tom"] = 'D';
16
17      std::cout << "Joe has a grade of " << grades["Joe"] << '\n';
18      std::cout << "Frank has a grade of " << grades["Frank"] << '\n';
19
20      return 0;
21  }
```

Prefer using `std::map` over writing your own implementation.

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

Question #3

Extra credit #2: Why doesn't this program work as expected?

```
1   #include <iostream>
2
3   int main()
4   {
5       GradeMap grades{};
6
```

```
7       char& gradeJoe{ grades["Joe"] }; // does a push_back
8       gradeJoe = 'A';
9
10      char& gradeFrank{ grades["Frank"] }; // does a push_back
11      gradeFrank = 'B';
12
13      std::cout << "Joe has a grade of " << gradeJoe << '\n';
14      std::cout << "Frank has a grade of " << gradeFrank << '\n';
15
16      return 0;
17  }
```

Show Solution

9.9 -- Overloading the parenthesis operator

Index

9.7 -- Overloading the increment and decrement operators

C++ TUTORIAL | PRINT THIS POST

191 comments to 9.8 — Overloading the subscript operator

« Older Comments 1 2 3

Ged
February 5, 2020 at 11:56 am · Reply

1. Is it possible to define std::vector size in a class?
When we write

```
int m_map[10]{};
```

It works, but if we try to do something like this.

```
std::vector<int> m_map(10);
```

We get an error.

2. Is it possible for the user to input the size of array?

For example

```
class Something
{
int* m_map = new int[UserEnteredSize];
```

3. When we write this line

```
private:
    std::vector<StudentGrade> m_map{};
```

How does it initialize? m_map[0] has both values of nothing and we do pushbacks that overwrite m_map[0] and add m_map[1] and later m_map[1] gets overwritten and m_map[2] gets added ?

nascardriver
February 6, 2020 at 9:19 am · Reply

1.
Members cannot be initialized with direct initialization (I don't know why).

```
std::vector<int> v{ std::vector<int>(10) };
```

Alternatively, use the member initializer list.

2.
Use a constructor with a member initializer list.

3.
Initialized to an empty vector. Vectors grow dynamically, see the lesson about vectors.

sito
January 24, 2020 at 1:02 pm · Reply

hello! for quiz 1c I decided to not try the find if function because it uses iterators and i don't fully understand how the function is supposed to work so I decided to see if a name exists in a vector by using a for each loop, looping through it and compare. My problem is though that despite overloading so that i compare struct one to struct 2, because the variable in the for each loop is a struct it won't work. Is this because I don't do the comparison in main?
here is my code below

```
// overloadingSubscriptOperatorQ1.cpp : This file contains the 'main' function. Program exe
//
#include <vector>
#include <iostream>
struct StudentGrade {
    std::string name;
    char grade;
};
class GradeMap {
private:
    std::vector <StudentGrade>m_map{};
```

```
12      StudentGrade student;
13  public:
14      char& operator[](std::string name) {
15          bool nameExists{ true };
16          for (const auto& i : m_map) {
17              if (i == student.name) {
18                  nameExists = true;
19                  return student.grade;
20              }
21          }
22          if (nameExists == false) {
23              m_map.pushback(name);
24              return m_map.back().grade;
25          }
26
27      }
28      friend bool operator==(const StudentGrade &student1, const StudentGrade& student2);
29  };
30
31  bool operator==(const StudentGrade& student1, const StudentGrade& student2) {
32      return student1.name == student2.name;
33  }
34
35  int main()
36  {
37      GradeMap grades{};
38      grades["Joe"] = 'A';
39      grades["Frank"] = 'B';
40      std::cout << "Joe has a grade of " << grades["Joe"] << '\n';
41      std::cout << "Frank has a grade of " << grades["Frank"] << '\n';
42      return 0;
43
44  }
```

nascardriver
January 25, 2020 at 2:05 am · Reply

`m_map` is a `std::vector` of `StudentGrade`. `i` (Line 16) is a `StudentGrade`, but `student.name` (Line 17) is a `std::string`. You can't compare a `StudentGrade` to a `std::string`. You need to compare the name of the student `i` to the name of `student`.

I added a simple sample implementation of `find` as a quiz to lesson P.6.8a. You were probably already past that lesson when I added it. Although iterators are unknown at that point, it might help you to understand how `find` is implemented with iterators (Pointers are basic iterators).

Attila
December 29, 2019 at 7:17 pm · Reply

Hello,

I feel like the quiz is very poorly paced.

1a) and b) are quite self-contained and almost trivial, and then 1c) is crammed with information.
I see, that the same was intended for 1c: "Write an overloaded operator[] for this class.", but it ended up a bit more complicated than that.

For the sake of explaining what the program should do, 1c should be split into two sections:
1c) for the for-each loop and it's purpuse
1d) for the rest of the overload.

1d) may be split further if necessary, but it defenetly needs to be rewritten. It's extremely confusing.

nascardriver
December 30, 2019 at 5:45 am · Reply

Hi Attila,

thanks for your feedback! We're currently writing new lessons about algorithms and lambdas, which will be useful in this quiz and reduce its length. I've marked this quiz for an update when the lessons are done.

vishs
October 30, 2019 at 11:29 pm · Reply

Hi,

While I agree that overloading [] operator is helpful for better comprehension and easier looking syntax in the examples you have taken above

But for other examples like:

```
class Database
{
    int list1[10];
    int list2[20];
    int someotherData;
};
```

Suppose I do this:

```
Database d1;
d1[2] = 3;
```

The above code is really not good, because class itself is having so many arrays and also non-array elements as members, in this case overloading as object[] doesn't give any idea of which list(list1 or list2) is being accessed and also what it means for "someotherData" member

So in reality, most classes will be having many data members of different types, so overloading operators with class objects gives no idea of what members are getting affected and in which way, so i feel overloading operators reduce readability of code? is my understanding correct?

nascardriver
October 31, 2019 at 4:21 am · Reply

Hi again,

operators, just like poorly named functions, can always be used in a way that makes them more confusing than helpful. You _can_ overload `operator[]` for `Database`, but as you said, it doesn't make a lot of sense. `operator[]` is used for map- and list-like objects.
Operators reduce code quality when they're used poorly, but increase it when used correctly. That's not a property unique to operators, you'll find it everywhere.

Guybrush87
September 27, 2019 at 6:19 am · Reply

Hello,

First of all, thank you for this great courses !

I have a silly question, why the code above don't compile with the error "error: invalid conversion from 'const int*' to 'int*' [-fpermissive]"

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

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

I don't modify nothing, why can I make my class member getList constant ?

Thank you for your help.

Guybrush87

nascardriver
September 27, 2019 at 6:27 am · Reply

`getList` is `const`.

```
const IntList list{};

list.getList()[0] = 123;
```

this would be legal if your code compiled, but obviously it shouldn't, because `list` is `const`.

Don't mark `getList` as `const` or return a `const int*`.

Guybrush87
September 30, 2019 at 12:16 am · Reply

Hello nascardriver,

I'm sorry, I don't understand your answer, here is my code :

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

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

int main()
{
    IntList list;

    return 0;
}
```

For this code the compiler says :
error: invalid conversion from 'const int*' to 'int*' [-fpermissive]
 int* getList() const { return m_list; }
                    ^~~~~~
In my code there is nothing const except for the function member getList.
What I don't understand is that my member getList doesn't modify any data member, why isn't it possible to declare it as a const member ?

Thank you very much for your help !

Guybrush87

nascardriver
September 30, 2019 at 2:41 am · Reply

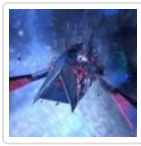
`getList` allows the caller to modify the object, because it returns a non-const pointer to a member variable. But you marked `getList` as `const`, which means that it's not possible to modify the members through that function.
That's a contradiction, so you get an error.

Guybrush87
September 30, 2019 at 4:28 am · Reply

Hello nascardriver,

Thank you !! I get it now.

Guybrush87

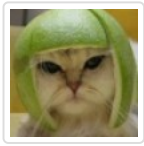


potterman28wxcv
September 15, 2019 at 8:22 am · Reply

Hello !

The 3) part scares me a lot about C++ ! Because from a glance, the code looks alright, and you just wouldn't see such a bug easily. Sorting out that bug requires having knowledge of how std::vector is working internally.. I guess std::vector is widely used, but if you use some library defined by another programmer and only have the API, there is no way you could easily trace back that bug.

Is there any general rule to follow in C++ to avoid writing the code of 3) ? That code really sends me shills because any reference could be "eventually dangling" as soon as you initialize them! I can't even imagine the amount of bugs that must have been caused by this kind of scenario in large projects..



Alex
September 19, 2019 at 11:55 am · Reply

As a general rule, we always have to be careful when holding references or pointers to data that lives in dynamic memory, because the destruction of that memory may not be predictable.

In this case, the scariness is mainly due to the fact that we don't expect operator[] to invalidate pointers or references, but it might. This is a function of using a std::vector to implement the class.

I can think of two possible solutions here:
1) Rewrite the interface to the class to make operations that might invalidate data always needs to be explicitly called by the user (e.g. have an add() function). At least that way, control of the lifetime of the data is under the user's control.
2) Better, use an implementation that doesn't invalidate the current set of elements when new ones are added. std::map doesn't invalidate references or pointers when operator[] is used with it (https://en.cppreference.com/w/cpp/container/map/operator_at), and so would be a better choice to implement this class with.

George
August 19, 2019 at 2:47 am · Reply

How are the constant and non-constant versions unique? Same return type and same number and type of arguments...

nascardriver
August 19, 2019 at 2:53 am · Reply

One is marked as `const`, the other one isn't.
The `const` version is used for `const` objects, and also for non-const objects if no non-const version exists. ie. the type of the object the function is called on determines which function gets called.

noobmaster
August 13, 2019 at 10:15 pm · Reply

I edited function char& GradeMap::operator[](const std::string &name) in solution for 1c

```
1  char& GradeMap::operator[](const std::string& name)
2  {
3      // See if we can find the name in the vector
4      for (auto &ref : m_map)
5      {
6          // If we found the name in the vector, return the grade
7          if (ref.name == name)
8              return ref.grade;
9          else // I added this
10         {
11             // otherwise create a new StudentGrade for this student
12             StudentGrade temp{ name, ' ' };
13
14             // otherwise add this to the end of our vector
15             m_map.push_back(temp);
16
17             // and return the element
18             return m_map.back().grade;
19         }
20     }
21 }
```

my compiler throws me a warning 'GradeMap::operator[]': not all control paths return a value
can someone explain why?

nascardriver
August 14, 2019 at 2:56 am · Reply

Your function has to return a value no matter what happens.
If `m_map` is empty, the loop is never entered and you don't return.
Your `else` can't be in the loop. You're only ever checking the first element.

noobmaster
August 15, 2019 at 11:29 pm · Reply

Got it. Thanks

mmp52
August 8, 2019 at 5:15 am · Reply

Hello!

in the for - each loop, what is the difference between making the iterator a reference or not? Also, if we use it like that how does compiler understand we did not mean the address but the iterated element's reference?

```
1
2 // See if we can find the name in the vector
3     for (auto &ref : m_map) //vs for (auto iteratedStudent : m_map)
```

thanks for the amazing content !!

nascardriver
August 8, 2019 at 6:24 am · Reply

Iterating over copies is slow. Non-fundamental types should always be passed/returned/iterated over by reference.
The first part of a for-each loop is a declaration, address-of doesn't make sense at that point, just like with regular variable declarations.

mmp52
August 8, 2019 at 7:19 am · Reply

Thank you!

Anastasia
August 28, 2019 at 11:39 am · Reply

But in this case using a copy shouldn't be an option at all, because this

```
1 // See if we can find the name in the vector
2 for (auto ref : m_map)
3 {
4     // If we found the name in the vector, return the grade
5     if (ref.name == name)
6     return ref.grade;
7 }
```

would return a reference to the local variable `ref` that's already out of scope! Am I wrong?

nascardriver
August 28, 2019 at 11:18 pm · Reply

You're right, I didn't see that!

Hai Linh
August 8, 2019 at 2:52 am · Reply

Quick question: Is there any reason to make index non-const? It would make sense to make index const, that way the function cannot change the value of index.

nascardriver
August 8, 2019 at 6:15 am · Reply

`index` is passed by copy, modifying it inside the operator won't affect the caller.
You can mark it as `const` if you like to, but it doesn't serve a practical reason.

Vir1oN
July 5, 2019 at 9:25 am · Reply

Could you refresh my memory, why do we use return by address instead of return by reference here?

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

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

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

    return 0;
}
```

nascardriver
July 5, 2019 at 9:31 am · Reply

References to arrays are ugly, it's easier to let them decay into pointers.

```
int (&getList())[10] { return m_list; }
```

Vir1oN
July 5, 2019 at 9:40 am · Reply

Uh, indeed. Thanks

Anthony
April 24, 2019 at 5:50 pm · Reply

Hi Alex,

Given the following class definition:

```
class GradeMap {
  private:
    struct StudentGrade {
        std::string name;
        char grade;
    };
    std::vector<StudentGrade> m_v {};
```

```
    public:
      char& operator[](const std::string str) {
          for (StudentGrade &sg : m_v) {
              if (sg.name == str)
                  return sg.grade;
          }
          StudentGrade tmp { str, 0 };
          m_v.push_back(tmp);
          return m_v.back().grade;
      }
};
```

The following works:

```
int main() {
    GradeMap a;
    GradeMap b(a);  // direct initialisation using the copy constructor
    return 0;
}
```

However, if I change

```
GradeMap a;
```

to

```
GradeMap a();
```

the line will compile, but

```
GradeMap b(a);
```

won't.

What is the difference between

```
GradeMap a;
```

and

```
GradeMap a();
```

? And why doesn't the line that follows compile?

Thanks!

nascardriver
April 26, 2019 at 2:43 am · Reply

```
GradeMap a();
```

is a function declaration. This is known as C++'s most vexing parse, because you might expect it to be a default initialization.
Brace initializers solve this problem.

```
GradeMap a{};
```

Anthony
April 23, 2019 at 8:55 am · Reply

This is a very interesting lesson. I initially omitted to use a reference within the for-each statement of question 1c, leaving a dangling reference. Code::Blocks picked it up with a warning about a local variable being returned. I'm learning :)

Tyson
April 18, 2019 at 4:12 am · Reply

Hi,

When I try to compile my solution or Alex' solution to Question-1C, I get this error "error: 'GradeMap::m_map' should be initialized in the member initialization list [-Werror=effc++]"

However, I was able to overcome it by adding empty curly brackets next to m_map, the private member, like this:

```
1 std::vector<StudentGrade> m_map {};
```

My question is, is this the correct way of solving the problem?

Thank you for the great tutorials.

nascardriver
April 18, 2019 at 5:08 am · Reply

yes

Alex
April 22, 2019 at 12:55 pm · Reply

This issue has been fixed in the lesson. Thanks for pointing out the omission.

Hamza
August 3, 2019 at 2:56 pm · Reply

I don't know why would it be a problem to leave it uninitialized .. I compiled it without curly brackets and it worked fine!

nascardriver
August 4, 2019 at 8:05 am · Reply

Not explicitly initializing it will still initialize it to an empty vector.
This only works, because `std::vector` a class-type and has automatic storage duration. If it was a fundamental type, you'd have to initialize it or risk undefined behavior on use. Since differentiating between the types (Initialize one but not the other) is extra work and could break if a type is changes in the future, initializing all types the same is best.

Louis Cloete
April 10, 2019 at 6:15 am · Reply

@Alex, I don't know if you know Rust, but I have started learning it as well. In Rust, Quiz q3 will not compile, because the compiler enforces that you have only one "mutable borrow" (non-const reference in C++) OR any amount of "immutable borrows" (const references in C++). This means you can't

mutate a vector while there are other references to it. This prevents dangling references/pointers and data races at compile time. Knowing this, I immediately spotted the error.

I find that my dabbling with Rust causes me to think safer in C++, because I had to fix many compiler errors because of code like this, which is legal in C++, but causes compiler errors in Rust.

« Older Comments 1 2 3