9.15 — Shallow vs. deep copying

BY ALEX ON NOVEMBER 9TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Shallow copying

Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides use a copying method known as a memberwise copy (also known as a **shallow copy**). This means that C++ copies each member of the class individually (using the assignment operator for overloaded operator=, and direct initialization for the copy constructor). When classes are simple (e.g. do not contain any dynamically allocated memory), this works very well.

For example, let's take a look at our Fraction class:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator{ numerator },
        m_denominator{ denominator }
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
}
```

The default copy constructor and assignment operator provided by the compiler for this class look something like this:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator{ numerator },
        m_denominator{ denominator }
    {
```

```
        assert(denominator != 0);
    }

    // Copy constructor
    Fraction(const Fraction &f) :
        m_numerator{ f.m_numerator },
        m_denominator{ f.m_denominator }
    {
    }

    Fraction& operator= (const Fraction &fraction);

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
}

// A better implementation of operator=
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // self-assignment guard
    if (this == &fraction)
        return *this;

    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

Note that because these default versions work just fine for copying this class, there's really no reason to write our own version of these functions in this case.

However, when designing classes that handle dynamically allocated memory, memberwise (shallow) copying can get us in a lot of trouble! This is because shallow copies of a pointer just copy the address of the pointer -- it does not allocate any memory or copy the contents being pointed to!

Let's take a look at an example of this:

```
#include <cstring> // for strlen()
#include <cassert> // for assert()

class MyString
{
private:
    char *m_data;
    int m_length;

public:
    MyString(const char *source="")
    {
        assert(source); // make sure source isn't a null string

        // Find the length of the string
        // Plus one character for a terminator
        m_length = std::strlen(source) + 1;
```

```
18
19          // Allocate a buffer equal to this length
20          m_data = new char[m_length];
21
22          // Copy the parameter string into our internal buffer
23          for (int i{ 0 }; i < m_length; ++i)
24              m_data[i] = source[i];
25
26          // Make sure the string is terminated
27          m_data[m_length-1] = '\0';
28      }
29
30      ~MyString() // destructor
31      {
32          // We need to deallocate our string
33          delete[] m_data;
34      }
35
36      char* getString() { return m_data; }
37      int getLength() { return m_length; }
38  };
```

The above is a simple string class that allocates memory to hold a string that we pass in. Note that we have not defined a copy constructor or overloaded assignment operator. Consequently, C++ will provide a default copy constructor and default assignment operator that do a shallow copy. The copy constructor will look something like this:

```
1  MyString::MyString(const MyString &source) :
2      m_length{ source.m_length },
3      m_data{ source.m_data }
4  {
5  }
```

Note that m_data is just a shallow pointer copy of source.m_data, meaning they now both point to the same thing.

Now, consider the following snippet of code:

```
1  int main()
2  {
3      MyString hello{ "Hello, world!" };
4      {
5          MyString copy{ hello }; // use default copy constructor
6      } // copy is a local variable, so it gets destroyed here.  The destructor deletes copy's s
7
8      std::cout << hello.getString() << '\n'; // this will have undefined behavior
9
10     return 0;
11 }
```

While this code looks harmless enough, it contains an insidious problem that will cause the program to crash! Can you spot it? Don't worry if you can't, it's rather subtle.

Let's break down this example line by line:

```
1      MyString hello{ "Hello, world!" };
```

This line is harmless enough. This calls the MyString constructor, which allocates some memory, sets hello.m_data to point to it, and then copies the string "Hello, world!" into it.

```
1      MyString copy{ hello }; // use default copy constructor
```

This line seems harmless enough as well, but it's actually the source of our problem! When this line is evaluated, C++ will use the default copy constructor (because we haven't provided our own). This copy constructor will do a shallow copy, initializing copy.m_data to the same address of hello.m_data. As a result, copy.m_data and hello.m_data are now both pointing to the same piece of memory!

```
} // copy gets destroyed here
```

When copy goes out of scope, the MyString destructor is called on copy. The destructor deletes the dynamically allocated memory that both copy.m_data and hello.m_data are pointing to! Consequently, by deleting copy, we've also (inadvertently) affected hello. Variable copy then gets destroyed, but hello.m_data is left pointing to the deleted (invalid) memory!

```
    std::cout << hello.getString() << '\n'; // this will have undefined behavior
```

Now you can see why this program has undefined behavior. We deleted the string that hello was pointing to, and now we are trying to print the value of memory that is no longer allocated.

The root of this problem is the shallow copy done by the copy constructor -- doing a shallow copy on pointer values in a copy constructor or overloaded assignment operator is almost always asking for trouble.

Deep copying

One answer to this problem is to do a deep copy on any non-null pointers being copied. A **deep copy** allocates memory for the copy and then copies the actual value, so that the copy lives in distinct memory from the source. This way, the copy and source are distinct and will not affect each other in any way. Doing deep copies requires that we write our own copy constructors and overloaded assignment operators.

Let's go ahead and show how this is done for our MyString class:

```
// assumes m_data is initialized
void MyString::deepCopy(const MyString& source)
{
    // first we need to deallocate any value that this string is holding!
    delete[] m_data;

    // because m_length is not a pointer, we can shallow copy it
    m_length = source.m_length;

    // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
    {
        // allocate memory for our copy
        m_data = new char[m_length];

        // do the copy
        for (int i{ 0 }; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = nullptr;
}

// Copy constructor
MyString::MyString(const MyString& source):
    m_data{ nullptr }
{
    deepCopy(source);
}
```

As you can see, this is quite a bit more involved than a simple shallow copy! First, we have to check to make sure source even has a string (line 11). If it does, then we allocate enough memory to hold a copy of that string (line 14).

Finally, we have to manually copy the string (lines 17 and 18).

Now let's do the overloaded assignment operator. The overloaded assignment operator is slightly trickier:

```
1  // Assignment operator
2  MyString& MyString::operator=(const MyString & source)
3  {
4      // check for self-assignment
5      if (this == &source)
6          return *this;
7
8      // now do the deep copy
9      deepCopy(source);
10
11     return *this;
12 }
```

Note that our assignment operator is very similar to our copy constructor, but there are three major differences:

- We added a self-assignment check.
- We return *this so we can chain the assignment operator.
- We need to explicitly deallocate any value that the string is already holding (so we don't have a memory leak when m_data is reallocated later).

When the overloaded assignment operator is called, the item being assigned to may already contain a previous value, which we need to make sure we clean up before we assign memory for new values. For non-dynamically allocated variables (which are a fixed size), we don't have to bother because the new value just overwrite the old one. However, for dynamically allocated variables, we need to explicitly deallocate any old memory before we allocate any new memory. If we don't, the code will not crash, but we will have a memory leak that will eat away our free memory every time we do an assignment!

A better solution

Classes in the standard library that deal with dynamic memory, such as std::string and std::vector, handle all of their memory management, and have overloaded copy constructors and assignment operators that do proper deep copying. So instead of doing your own memory management, you can just initialize or assign them like normal fundamental variables! That makes these classes simpler to use, less error-prone, and you don't have to spend time writing your own overloaded functions!

Summary

- The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.
- Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.
- Favor using classes in the standard library over doing your own memory management.

9.x -- Chapter 9 comprehensive quiz

Index

9.14 -- Overloading the assignment operator

C++ TUTORIAL | PRINT THIS POST

95 comments to 9.15 — Shallow vs. deep copying

« Older Comments 1 2

sito
February 7, 2020 at 6:57 am · Reply

in this lesson for the MyString class why do you have the char as const in the constructor? I tried without const and i got a convertion error.

nascardriver
February 7, 2020 at 9:01 am · Reply

A string literal can't be stored in a `char*`.
Whenever you have a pointer or reference and don't modify it, mark it `const`.

inspectorPlatypus
December 21, 2019 at 12:15 pm · Reply

```
// self-assignment guard
    if (this == &fraction)
return *this;
```

Hey Alex, can you clarify what these lines are doing i have a bit of trouble understanding them.

nascardriver
December 22, 2019 at 2:51 am · Reply

It checks if `fraction` is the same object as `this`. Not comparing the contents of the fractions, but comparing their addresses. This is covered in lesson 9.14.

mmp52

August 8, 2019 at 8:07 am · Reply

Hello,

in the below snippet of yours, you delete the m_data at 5th line of the MyString::deepCopy(..) function, but still when you define copy constructor you assign m_data(nullptr) at 26th line knowing that it will call deepCopy and be deleted , why is that?

```
1  // assumes m_data is initialized
2  void MyString::deepCopy(const MyString& source)
3  {
4      // first we need to deallocate any value that this string is holding!
5      delete[] m_data;
6
7      // because m_length is not a pointer, we can shallow copy it
8      m_length = source.m_length;
9
10     // m_data is a pointer, so we need to deep copy it if it is non-null
11     if (source.m_data)
12     {
13         // allocate memory for our copy
14         m_data = new char[m_length];
15
16         // do the copy
17         for (int i=0; i < m_length; ++i)
18             m_data[i] = source.m_data[i];
19     }
20     else
21         m_data = nullptr;
22 }
23
24 // Copy constructor
25 MyString::MyString(const MyString& source):
26     m_data(nullptr)
27 {
28     deepCopy(source);
29 }
```

thanks!

nascardriver

August 8, 2019 at 8:14 am · Reply

All members should be initialized during construction.
`delete[]` doesn't modify the pointer, it deletes the pointed-to object. Since `nullptr` doesn't point anywhere, `delete[]` has no effect.
If `m_data` wasn't initialized, `delete[]` would try to delete an invalid pointer, causing undefined behavior.

mmp52

August 27, 2019 at 11:19 pm · Reply

thank you

Sergey

September 8, 2019 at 8:37 am · Reply

-"All members should be initialized during construction."
But why the m_length is not initialized on 26th line then?

nascardriver
September 8, 2019 at 8:40 am · Reply

`deepCopy` is called by that constructor. `deepCopy` overrides `m_length` and doesn't read from it, so it doesn't have to be initialized. `m_data` has to be initialized, because `deepCopy` reads from it.

hersel99
November 12, 2019 at 8:42 am · Reply

Also, I assume, if only used in constructor copy, line 11 `delete[]` of `deepCopy` makes no senses because a new object is created, but because we our overloading the assignment operator(=) (and reusing `deepCopy`) when our object could already be created (and having m_data allocated) we do `delete[]`.

Arthur
August 5, 2019 at 1:46 pm · Reply

Hi. Some typos after first code snippet showing deep copying.
As you can see, this is quite a bit more involved than a simple shallow copy! First, we have to check to make sure source even has a string (line 8(must be 11)). If it does, then we allocate enough memory to hold a copy of that string (line 11(must be 14)). Finally, we have to manually copy the string (lines 14 and 15(must be 17 and 18)).

Ejamesr
November 11, 2019 at 7:59 am · Reply

Alex, the typos Arthur pointed out still remain to be fixed (i.e., the line numbers referenced in your description need to be adjusted).

And as so many others have said, thank you for such a finely detailed, logical, and orderly overview of C++!

nascardriver
November 11, 2019 at 8:13 am · Reply

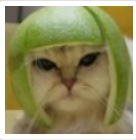
Thanks for the reminder! Lesson updated.

DecSco
July 31, 2019 at 8:51 am · Reply

Wouldn't it be preferable to put the deep copying logic into a function to avoid code duplication?

```
1  void MyString::deepCopy( const MyString& source )
2  {
3      // because m_length is not a pointer, we can shallow copy it
4      m_length = source.m_length;
5
6      // m_data is a pointer, so we need to deep copy it if it is non-null
7      if (source.m_data)
8      {
9          // allocate memory for our copy
```

```
        m_data = new char[m_length];

        // do the copy
        for (int i=0; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = nullptr;
}
```



Alex
August 3, 2019 at 9:05 pm · Reply

Yes, though deepCopy should handle an already allocated m_data -- otherwise calling the function could cause the original m_data to be leaked. I've updated the lesson accordingly.

Paulo Filipe
June 6, 2019 at 5:37 am · Reply

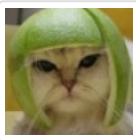
In this code snippet present above in this lesson:

```
// Copy constructor
MyString::MyString(const MyString& source)
{
    // because m_length is not a pointer, we can shallow copy it
    m_length = source.m_length;

    // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
    {
        // allocate memory for our copy
        m_data = new char[m_length];

        // do the copy
        for (int i=0; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = 0;
}
```

Shouldn't line 18 be:

```
m_data = nullptr;
```

instead?



Alex
June 12, 2019 at 2:47 pm · Reply

Yes. Updated. Thanks!

Chandra Shekhar
April 16, 2019 at 8:21 pm · Reply

MyString::MyString(const MyString& source)
{

```
    // because m_length is not a pointer, we can shallow copy it
    m_length = source.m_length;

    // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
    {
        // allocate memory for our copy
        m_data = new char[m_length];

        // do the copy
        for (int i=0; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = 0;
}
```

In the above code for deep copy
The m_length = source.m_length should have been m_length = source.m_length+1 to allocate one extra bit to store the bull character '\0' Right????

nascardriver
April 17, 2019 at 2:59 am · Reply

No, @source.m_length includes the 0-terminator already.

Richard
March 5, 2019 at 1:01 pm · Reply

Std::vector tends to be a little slow and std::array has some of the limitations of C-style arrays. While C++ standard library provides tools of first choice, occasionally a class needs to perform its own dynamic memory management.

Here's the skeleton of a container class that encapsulates a dynamic array and defines its special member functions: an all-in-one illustration of tutorial concepts up to "Inheritance". To keep it simple, I've substituted a global const MAX for a potential member variable, m_length. The latter would be required for flexible-sized arrays.

To demonstrate how the compiler handles the move constructor and move assignment class functions, I've inserted calls to a function printAp(), which display the addresses of selected objects. Calls to printAp are bracketed in the class functions to mark them for later removal. To force the creation of an anonymous (temporary) object, a member function reverse() returns an object of the class. Main() serves as a test program as discussed below:

```
1  #include <iostream>
2  #include <string>
3
4  using byte = unsigned char;
5  constexpr int MAX{ 10 };
6
7  class Array;
8  void printAp(std::string source, Array *a_ptr)        // print addy of an Array obj
9  {
10     std::cout << source << " address = " << a_ptr << '\n';
11 }
12
13 class Array {
14 private:
15     byte *m_ptr;
```

```
public:
    Array()  { m_ptr = new byte[MAX]{}; }        // default constructor
    ~Array() { delete[] m_ptr; }             // destructor
    Array(const Array &rhs)                 // copy constructor
    {
        if (rhs.m_ptr != nullptr) {
            m_ptr = new byte[MAX];
            // pointer arithmetic avoids lvl 4 warnings:
            for (int i = 0; i < MAX; i++) *(m_ptr + i) = *(rhs.m_ptr + i);
        }
        else m_ptr = nullptr;
    }
    Array(Array &&rhs) : m_ptr(rhs.m_ptr)        // move constructor
    {
        {  printAp("in move construc, rhs", &rhs);  }
        rhs.m_ptr = nullptr;
        {  printAp("in move construc, this", this); }
    }
    Array& operator=(const Array &rhs)        // copy assignment (with chaining)
    {
        if (this != &rhs) {
            delete[] m_ptr;
            if (rhs.m_ptr != nullptr) {
                m_ptr = new byte[MAX];
                for (int i = 0; i < MAX; i++) *(m_ptr + i) = *(rhs.m_ptr + i);
            }
            else m_ptr = nullptr;
        }
        return *this;
    }
    Array& operator=(Array &&rhs)            // move assignment with chaining
    {
        {  printAp("in move assign, rhs", &rhs);   }
        if (this != &rhs) {            // self-assign really possible here?
            delete[] m_ptr;
            m_ptr = rhs.m_ptr;
            rhs.m_ptr = nullptr;
        }
        {  printAp("in move assign, this", this);  }
        return *this;
    }
    void setElem(const int index, const byte val)    // set an elem (no range check)
    {
        *(m_ptr + index) = val;
    }
    byte getElem(const int index) const { return *(m_ptr + index); } // get an elem

    // print elem with comment txt
    void printCElem(const std::string comment, const int index) const
    {
      std::cout << comment << ", " << static_cast<int>(this->getElem(index)) << "\n\n";
    }
    Array reverse() const                // create new obj with elem's reversed
    {
        Array Temp;
        for (int i = 0; i < MAX; i++) *(Temp.m_ptr + MAX - i - 1) = *(m_ptr + i);
        {  printAp("in reverse, Temp", &Temp);    }
        return Temp;
    }
};

int main()
```

```
{
    Array A;                    // default constructor
    A.printCElem("Default Construc", 0);        // show initialization

    A.setElem(0, 0x01);
    Array B = A;                    // copy constructor
    B.printCElem("Copy Construc", 0);

    Array C;
    C.setElem(0, 0x03);
    A = B = C;                    // copy assignment with chaining
    B.printCElem("First Assign", 0);
    A.printCElem("Chain Assign", 0);

    C.setElem(MAX-1, 0x05);
    Array D = C.reverse();                // move constructor
    D.printCElem("Move Construc", 0);

    B = D.reverse();                // move assignment
    B.printCElem("Move Assign", 0);

    printAp("object B", &B);
    printAp("object C", &C);
    printAp("object D", &D);

    system("pause");
    return 0;
}
```

Main does 2 things. Superficially, it creates objects A, B, C, and D. These objects encapsulate unsigned char arrays. Main sets and retrieves values of the first or last elements of the arrays during the test of each class member function. On a deeper level, main shows the behavior of the move constructor and move assignment member functions using the "print a pointer" function, printAp(). The console output on my machine using Visual Studio 2017 in DEBUG mode:

Default Construc, 0

Copy Construc, 1

First Assign, 3

Chain Assign, 3

in reverse, Temp address = 0041F5E4
in move construc, rhs address = 0041F5E4
in move construc, this address = 0041F774
Move Construc, 5

in reverse, Temp address = 0041F5E4
in move construc, rhs address = 0041F5E4
in move construc, this address = 0041F66C
in move assign, rhs address = 0041F66C
in move assign, this address = 0041F78C
Move Assign, 3

object B address = 0041F78C
object C address = 0041F780
object D address = 0041F774

This output is far simpler than it first appears. There's text to show the purpose of each test, and to print an element of the encapsulated array to verify the result. So far so dull.

The rest is interesting. The third-to-last block of text shows a call to my move constructor for the statement D = C.reverse(). What I want to do is create D and move the result of reverse() into it. To accomplish this, the compiler calls my move constructor for D and hands me an r-value reference to the object "Temp", which I created in reverse() for a return value. Smart! Within the move constructor, "this" contains the address of D, and &rhs contains the address of "Temp". As Temp is already constructed, all I really need to do is "capture" its data. My move constructor accomplishes this task by first assigning Temp's m_ptr to D's, then assigning NULL to Temp's m_ptr (effectively marking the latter unusable upon its release to the heap). Voila! So far so good.

But what's all this mess for B = D.reverse()? B already exists, so what I want to do is move the result of reverse() directly to it. The compiler does this in 2 steps. First, it creates an anonymous object, then it calls my move constructor to capture Temp's data to it. Just like it did for D above. What? Alright, we get it: move constructor is named a "constructor" for a reason. After capturing "Temp" as an anonymous object, the compiler calls my move assignment function, for the purpose of moving the contents of its anonymous object into my preexisting variable B. Notice in the move assignment function, the anonymous object is passed as an r-value reference, and you can trace its address from "this" (in the move constructor) to &rhs (a parameter in move assignment). Now, within move assignment, "this" points to my existing object, B. The contents of object B will be replaced, so I delete[] its dynamic memory. Then another rehash: I "capture" the anonymous object's data by first assigning the memory ptr of the anonymous object to B's m_ptr, followed by assigning NULL to rhs.m_ptr (marking the latter as unusable upon its release to the heap).

I'd prefer doing the B = D.reverse() in one move-assignment step. Here's the output from the RELEASE build with optimization, skipping up to "Chain Assign":

```
Chain Assign, 3

in reverse, Temp address = 003AF9DC
Move Construc, 5

in reverse, Temp address = 003AF9D8
in move assign, rhs address = 003AF9D8
in move assign, this address = 003AF9E8
Move Assign, 3

object B address = 003AF9E8
object C address = 003AF9E4
object D address = 003AF9DC
```

The optimized .exe fulfills my wish: it calls my move assignment function with "this" pointing to B, while passing an r-value reference to "Temp" directly. What about my D = C.reverse()? My move constructor isn't even called! D is simply given Temp's address. Hurray for elision!

The reason for looking "under-the-hood" of this process is to know what demands Array objects place on the heap. reverse() passes its return "by-value", but it does not consume any significant heap storage beyond what is required to generate an object for the return (in my case, "Temp"). This understanding is important when MAX is expanded to the thousands.

Alex makes this clear: as a member variable of Array class, m_ptr keeps the code RAII compliant, so (hopefully) no dangling pointer runs amuck. The big disadvantage of doing one's own memory management is that less-than-meticulous code (like not checking ranges) can corrupt the heap.

Asgar
February 19, 2019 at 9:21 am · Reply

Hi.

About this line in the first paragraphs:
"... C++ copies each member of the class individually (using the assignment operator for overloaded operator=, and direct initialization for the copy constructor)."

Did you mean to say "initialization" rather than "direct initialization"? As I understand, a copy constructor is invoked during any of the 3 kinds of initialization:
1. copy initialization,
2. direct initialization, and
3. uniform initialization

Alex
February 19, 2019 at 8:34 pm · Reply

In this context, we're talking about copying the _members_ of a class. These members are initialized using direct initialization (as opposed to copy initialization).

Jon
January 29, 2019 at 3:18 am · Reply

Hello again, I can't quite figure something out - for the following code snippet you wrote:

```
1  int main()
2  {
3      MyString hello("Hello, world!");
4      {
5          MyString copy = hello; // use default copy constructor
6      } // copy is a local variable, so it gets destroyed here.  The destructor deletes copy'
7
8      std::cout << hello.getString() << '\n'; // this will have undefined behavior
9
10     return 0;
11 }
```

If I delete the brackets on lines 4 and 6, the program compiles, but I get the runtime error:
Test(50460,0x10039b380) malloc: *** error for object 0x100403700: pointer being freed was not allocated.

I was under the impression that this would run OK because the variables would all stay in scope until the end of the program and because deleting a null pointer has no effect (is that not what this error message is pointing to?) but it looks like I'm missing something. It does print out "Hello World!" but something is not quite right...

Jon
January 29, 2019 at 3:45 am · Reply

Think I've got it, I believe I conflated null pointer and uninitialized pointer?

nascardriver
January 29, 2019 at 8:48 am · Reply

All pointers in this example have been initialized, neither is a nullptr.
Both @hello and @copy point to the same char array. Once one of @hello or @copy goes out of scope, the char array will be deleted. The other variable will still point to were the char array used to be. When the other variable goes out of scope, the char array will be deleted again, but it doesn't exist. Behavior is undefined.

nascardriver
August 23, 2018 at 9:22 am · Reply

Hi Alex!

Third code block, line 17 should use @std::strlen

Alex
August 25, 2018 at 8:34 am · Reply

Fixed! Thanks!

seriouslysupersonic
August 23, 2018 at 9:15 am · Reply

Hi!

1) Since

```
1  MyString(const char *source="")
2      {
3          assert(source); // make sure source isn't a null string
```

aren't we sure that we will never be copying / assigning a non-null string (if the allocation was successful)?

2) Shouldn't we assign @m_length after allocating new memory and checking @m_data is non-null?

3) Is there a reason not to use strcpy() and use a for loop?

nascardriver
August 23, 2018 at 9:19 am · Reply

Hi!

1)

```
1  MyString(nullptr);
```

2)
The order doesn't matter

3)
No

seriouslysupersonic
August 23, 2018 at 9:40 am · Reply

Thank you for the quick reply.

1) If we do that, wouldn't the assertion fail and we wouldn't be able to copy / assign a null source.m_data field - because (unless memory allocation failed) no object with a null m_data field would be created? Shouldn't we check instead that new didn't fail?

2) But if @source.m_data is null because the memory allocation of the source object failed, we are assigning whatever length the source believes it successfully allocated when the source object was created while we assign a nullptr to @m_data.

nascardriver
August 23, 2018 at 9:53 am · Reply

1) There is not @source object, @source is a const char*. I might have misunderstood your question. Can you elaborate on (1)?
> Shouldn't we check instead that new didn't fail?
All we can do is check for nullptr. If allocation fails without the noexcept version of @new, an exception is thrown and this constructor won't be reached anyway.

2) If @source failed to allocate memory, it will have thrown an exception and you shouldn't be able to use the source object anymore. But I agree with you, setting the length after verifying that there actually is data is better.

seriouslysupersonic
August 23, 2018 at 10:08 am · Reply

1) I think I wasn't too clear in the beginning (sorry for my English). What I meant was: every MyString object seems to have a non-null m_data field because we either assert that in the const char* constructor or, as you just explained, if new fails, the constructor won't be reached anyway. If that's true, why do we then check

```
1 if (source.m_data)
```

in the copy constructor and assignment operator overload?

2) This question doesn't make much sense because I forgot new would throw an exception and then we would have to handle that. Thanks for the explanation!

nascardriver
August 23, 2018 at 10:16 am · Reply

> why do we then check
Seems redundant. But if you were to add another constructor or function that allows @m_data to be a nullptr, this constructor wouldn't break because of that update.

« Older Comments 1 2