13.16 — Shallow vs. deep copying

ALEX NOVEMBER 13, 2021

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:

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

class Fraction
{
private:
    int m_numerator { 0 };
    int m_denominator { 1 };

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 default assignment operator provided by the compiler for this class look something like this:

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

class Fraction
{
private:
    int m_numerator { 0 };
    int m_denominator { 1 };

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

    // Possible implementation of implicit copy constructor
    Fraction(const Fraction& f)
        : m_numerator( f.m_numerator )
        , m_denominator( f.m_denominator )
    {
    }

    // Possible implementation of implicit assignment operator
    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;
    }

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

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:

```cpp
#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;

        // Allocate a buffer equal to this length
        m_data = new char[m_length];

        // Copy the parameter string into our internal buffer
        for (int i{ 0 }; i < m_length; ++i)
            m_data[i] = source[i];
    }

    ~MyString() // destructor
    {
        // We need to deallocate our string
        delete[] m_data;
    }

    char* getString() { return m_data; }
    int getLength() { return m_length; }
};
```

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:

MyString(hello)

```cpp
MyString::MyString(const MyString& source)
    : m_length( source.m_length )
    , m_data( source.m_data )          默认的复制构造函数(做浅拷贝)
{
}
```

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:

```cpp
int main()
{
    MyString hello{ "Hello, world!" };
    {
        MyString copy{ hello }; // use default copy constructor
    } // copy is a local variable, so it gets destroyed here.  The destructor deletes copy's string, which leaves hello with a dangling
pointer

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

    return 0;
}
```

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:

```cpp
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.

```cpp
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!

指针成员的浅复制

```cpp
} // 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!

```cpp
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:

```cpp
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 {
27     deepCopy(source);
28 }
```

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:

```cpp
// Assignment operator
MyString& MyString::operator=(const MyString& source)
{
    // check for self-assignment
    if (this != &source)
    {
        // now do the deep copy
        deepCopy(source);
    }

    return *this;
}
```

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.

Next lesson
13.17 Overloading operators and function templates

Back to table of contents

Previous lesson
13.15 Overloading the assignment operator