What is the copy-and-swap idiom?

What is this idiom and when should it be used? Which problems does it solve? Does the idiom change when C++11 is used?

Although it's been mentioned in many places, we didn't have any singular "what is it" question and answer, so here it is. Here is a partial list of places where it was previously mentioned:

- What are your favorite C++ Coding Style idioms: Copy-swap
- Copy constructor and = operator overload in C++: is a common function possible?
- What is copy elision and how it optimizes copy-and-swap idiom
- C++: dynamically allocating an array of objects?

c++ copy-constructor assignment-operator c++-faq copy-and-swap

edited May 6 '13 at 6:50
Joe Gauterin
10.3k ● 21 ● 54

asked Jul 19 '10 at 8:42
GManNickG
204k ● 24 ● 321 ● 456

3 gotw.ca/gotw/059.htm from Herb Sutter – DumbCoder Jul 19 '10 at 8:51

7 good question & great answer, but should probably be community wiki? – Philipp Jul 19 '10 at 8:54

26 @Philipp, as more contributions will be edited in by users, it will eventually *become* a community wiki. But until that--why wouldn't we let GMan earn some reputation for his work? – Pavel Shved Jul 19 '10 at 9:01

1 @DumbCoder: Ah, I forgot about that, thanks. @Philipp: Arguable. :) Just don't up-vote the question, I suppose, but although I understand the idea it's still just a question and answer. It just happened to be the same person. (Obviously if I did this everyday it would be a problem.) – GManNickG Jul 19 '10 at 9:01

3 Good idea to have a full-fledge explanation for this idiom, it's so common that everyone should know about it. – Matthieu M. Jul 19 '10 at 11:52

5 Answers

Overview

Why do we need the copy-and-swap idiom?

Any class that manages a resource (a *wrapper*, like a smart pointer) needs to implement The Big Three. While the goals and implementation of the copy-constructor and destructor are straightforward, the copy-assignment operator is arguably the most nuanced and difficult. How should it be done? What pitfalls need to be avoided?

The *copy-and-swap idiom* is the solution, and elegantly assists the assignment operator in achieving two things: avoiding code duplication, and providing a strong exception guarantee.

How does it work?

Conceptually, it works by using the copy-constructor's functionality to create a local copy of the data, then takes the copied data with a `swap` function, swapping the old data with the new data. The temporary copy then destructs, taking the old data with it. We are left with a copy of the new data.

In order to use the copy-and-swap idiom, we need three things: a working copy-constructor, a working destructor (both are the basis of any wrapper, so should be complete anyway), and a `swap` function.

A swap function is a *non-throwing* function that swaps two objects of a class, member for member. We might be tempted to use `std::swap` instead of providing our own, but this would be impossible; `std::swap` uses the copy-constructor and copy-assignment operator within its implementation, and we'd ultimately be trying to define the assignment operator in terms of itself!

(Not only that, but unqualified calls to `swap` will use our custom swap operator, skipping over the unnecessary construction and destruction of our class that `std::swap` would entail.)

An in-depth explanation

The goal

Let's consider a concrete case. We want to manage, in an otherwise useless class, a dynamic array. We start with a working constructor, copy-constructor, and destructor:

```
#include <algorithm> // std::copy
#include <cstddef> // std::size_t

class dumb_array
{
public:
    // (default) constructor
    dumb_array(std::size_t size = 0)
        : mSize(size),
          mArray(mSize ? new int[mSize]() : 0)
    {
    }

    // copy-constructor
    dumb_array(const dumb_array& other)
        : mSize(other.mSize),
          mArray(mSize ? new int[mSize] : 0),
    {
        // note that this is non-throwing, because of the data
        // types being used; more attention to detail with regards
        // to exceptions must be given in a more general case, however
        std::copy(other.mArray, other.mArray + mSize, mArray);
    }

    // destructor
    ~dumb_array()
    {
        delete [] mArray;
    }

private:
    std::size_t mSize;
    int* mArray;
};
```

This class almost manages the array successfully, but it needs `operator=` to work correctly.

A failed solution

Here's how a naive implementation might look:

```
// the hard part
dumb_array& operator=(const dumb_array& other)
{
    if (this != &other) // (1)
    {
        // get rid of the old data...
        delete [] mArray; // (2)
        mArray = 0; // (2) *(see footnote for rationale)

        // ...and put in the new
        mSize = other.mSize; // (3)
        mArray = mSize ? new int[mSize] : 0; // (3)
        std::copy(other.mArray, other.mArray + mSize, mArray); // (3)
    }

    return *this;
}
```

And we say we're finished; this now manages an array, without leaks. However, it suffers from three problems, marked sequentially in the code as `(n)`.

1. The first is the self-assignment test. This check serves two purposes: it's an easy way to prevent us from running needless code on self-assignment, and it protects us from subtle bugs (such as deleting the array only to try and copy it). But in all other cases it merely serves to slow the program down, and act as noise in the code; self-assignment rarely occurs, so most of the time this check is a waste. It would be better if the operator could work properly without it.

2. The second is that it only provides a basic exception guarantee. If `new int[mSize]` fails, `*this` will have been modified. (Namely, the size is wrong and the data is gone!) For a strong exception guarantee, it would need to be something akin to:

   ```
   dumb_array& operator=(const dumb_array& other)
   ```

```
{
    if (this != &other) // (1)
    {
        // get the new data ready before we replace the old
        std::size_t newSize = other.mSize;
        int* newArray = newSize ? new int[newSize]() : 0; // (3)
        std::copy(other.mArray, other.mArray + newSize, newArray); // (3)

        // replace the old data (all are non-throwing)
        delete [] mArray;
        mSize = newSize;
        mArray = newArray;
    }

    return *this;
}
```

3. The code has expanded! Which leads us to the third problem: code duplication. Our assignment operator effectively duplicates all the code we've already written elsewhere, and that's a terrible thing.

In our case, the core of it is only two lines (the allocation and the copy), but with more complex resources this code bloat can be quite a hassle. We should strive to never repeat ourselves.

(One might wonder: if this much code is needed to manage one resource correctly, what if my class manages more than one? While this may seem to be a valid concern, and indeed it requires non-trivial `try` / `catch` clauses, this is a non-issue. That's because a class should manage *one resource only*!)

A successful solution

As mentioned, the copy-and-swap idiom will fix all these issues. But right now, we have all the requirements except one: a `swap` function. While The Rule of Three successfully entails the existence of our copy-constructor, assignment operator, and destructor, it should really be called "The Big Three and A Half": any time your class manages a resource it also makes sense to provide a `swap` function.

We need to add swap functionality to our class, and we do that as follows†:

```
class dumb_array
{
public:
    // ...

    friend void swap(dumb_array& first, dumb_array& second) // nothrow
    {
        // enable ADL (not necessary in our case, but good practice)
        using std::swap;

        // by swapping the members of two classes,
        // the two classes are effectively swapped
        swap(first.mSize, second.mSize);
        swap(first.mArray, second.mArray);
    }

    // ...
};
```

Now not only can we swap our `dumb_array`'s, but swaps in general can be more efficient; it merely swaps pointers and sizes, rather than allocating and copying entire arrays. Aside from this bonus in functionality and efficiency, we are now ready to implement the copy-and-swap idiom.

Without further ado, our assignment operator is:

```
dumb_array& operator=(dumb_array other) // (1)
{
    swap(*this, other); // (2)

    return *this;
}
```

And that's it! With one fell swoop, all three problems are elegantly tackled at once.

Why does it work?

We first notice an important choice: the parameter argument is taken *by-value*. While one could just as easily do the following (and indeed, many naive implementations of the idiom do):

```
dumb_array& operator=(const dumb_array& other)
{
    dumb_array temp(other);
    swap(*this, temp);

    return *this;
}
```

We lose an important optimization opportunity. Not only that, but this choice is critical in C++11, which is discussed later. (On a general note, a remarkably useful guideline is as follows: if you're going to make a copy of something in a function, let the compiler do it in the

parameter list.‡)

Either way, this method of obtaining our resource is the key to eliminating code duplication: we get to use the code from the copy-constructor to make the copy, and never need to repeat any bit of it. Now that the copy is made, we are ready to swap.

Observe that upon entering the function that all the new data is already allocated, copied, and ready to be used. This is what gives us a strong exception guarantee for free: we won't even enter the function if construction of the copy fails, and it's therefore not possible to alter the state of `*this`. (What we did manually before for a strong exception guarantee, the compiler is doing for us now; how kind.)

At this point we are home-free, because `swap` is non-throwing. We swap our current data with the copied data, safely altering our state, and the old data gets put into the temporary. The old data is then released when the function returns. (Where upon the parameter's scope ends and its destructor is called.)

Because the idiom repeats no code, we cannot introduce bugs within the operator. Note that this means we are rid of the need for a self-assignment check, allowing a single uniform implementation of `operator=`. (Additionally, we no longer have a performance penalty on non-self-assignments.)

And that is the copy-and-swap idiom.

What about C++11?

The next version of C++, C++11, makes one very important change to how we manage resources: the Rule of Three is now **The Rule of Four** (and a half). Why? Because not only do we need to be able to copy-construct our resource, we need to move-construct it as well.

Luckily for us, this is easy:

```
class dumb_array
{
public:
    // ...

    // move constructor
    dumb_array(dumb_array&& other)
        : dumb_array() // initialize via default constructor, C++11 only
    {
        swap(*this, other);
    }

    // ...
};
```

What's going on here? Recall the goal of move-construction: to take the resources from another instance of the class, leaving it in a state guaranteed to be assignable and destructible.

So what we've done is simple: initialize via the default constructor (a C++11 feature), then swap with `other`; we know a default constructed instance of our class can safely be assigned and destructed, so we know `other` will be able to do the same, after swapping.

(Note that some compilers do not support constructor delegation; in this case, we have to manually default construct the class. This is an unfortunate but luckily trivial task.)

Why does that work?

That is the only change we need to make to our class, so why does it work? Remember the ever-important decision we made to make the parameter a value and not a reference:

```
dumb_array& operator=(dumb_array other); // (1)
```

Now, if `other` is being initialized with an rvalue, *it will be move-constructed*. Perfect. In the same way C++03 let us re-use our copy-constructor functionality by taking the argument by-value, C++11 will *automatically* pick the move-constructor when appropriate as well. (And, of course, as mentioned in previously linked article, the copying/moving of the value may simply be elided altogether.)

And so concludes the copy-and-swap idiom.

Footnotes

*Why do we set `mArray` to null? Because if any further code in the operator throws, the destructor of `dumb_array` might be called; and if that happens without setting it to null, we attempt to delete memory that's already been deleted! We avoid this by setting it to null, as deleting null is a no-operation.

†There are other claims that we should specialize `std::swap` for our type, provide an in-class `swap` along-side a free-function `swap`, etc. But this is all unnecessary: any proper use of `swap` will be through an unqualified call, and our function will be found through ADL. One function will do.

‡The reason is simple: once you have the resource to yourself, you may swap and/or move it (C++11) anywhere it needs to be. And by making the copy in the parameter list, you maximize optimization.

edited Nov 6 '15 at 23:39

wil93
876 ● 6 ● 18

answered Jul 19 '10 at 8:43

GManNickG
204k ● 24 ● 321 ● 456

6 Also note that copy-and-swap also needs a dtor. (Actually that goes without saying, but since you mention the cctor, you might as well mention the dtor.) – sbi Jul 19 '10 at 9:04

14 @GMan: I would argue that a class managing several resources at once is doomed to fail (exception safety becomes nightmarish) and I would strongly recommend that either a class manages ONE resource OR it has business functionality and use managers. – Matthieu M. Jul 19 '10 at 11:53

4 @FrEEzE: "It's compiler specific in which order the list is processed." No it's not. It's processed in the order they appear in the class definition. A compiler that doesn't accept `std::copy` that way is broken, I'm not coding for broken compilers. And I'm not sure I understand your last comment. – GManNickG Jun 21 '11 at 16:05

9 I don't get why swap method is declared as friend here? – szx Dec 13 '11 at 6:10

6 @neuviemeporte: With the parenthesis, the arrays elements are default initialized. Without, they are uninitialized. Since in the copy constructor we'll be overwriting the values anyway, we can skip initialization. – GManNickG Jul 19 '12 at 15:01

Assignment, at its heart, is two steps: ***tearing down the object's old state*** and ***building its new state as a copy*** of some other object's state.

Basically, that's what the ***destructor*** and the ***copy constructor*** do, so the first idea would be to delegate the work to them. However, since destruction mustn't fail, while construction might, *we actually want to do it the other way around*: ***first perform the constructive part*** and if that succeeded, ***then do the destructive part***. The copy-and-swap idiom is a way to do just that: It first calls a class' copy constructor to create a temporary, then swaps its data with the temporary's, and then lets the temporary's destructor destroy the old state.
Since `swap()` is supposed to never fail, the only part which might fail is the copy-construction. That is performed first, and if it fails, nothing will be changed in the targeted object.

In its refined form, copy-and-swap is implemented by having the copy performed by initializing the (non-reference) parameter of the assignment operator:

```
T& operator=(T tmp)
{
    this->swap(tmp);
    return *this;
}
```

edited Dec 22 '10 at 14:26

answered Jul 19 '10 at 8:55

sbi
121k ● 31 ● 162 ● 328

I think that mentioning the pimpl is as important as mentioning the copy, the swap and the destruction. The swap isn't magically exception-safe. It's exception-safe because swapping pointers is exception-safe. You don't **have** to use a pimpl, but if you don't then you must make sure that each swap of a member is exception-safe. That can be a nightmare when these members can change and it is trivial when they're hidden behind a pimpl. And then, then comes the cost of the pimpl. Which leads us to the conclusion that often exception-safety bears a cost in performance. – wilhelmtell Dec 22 '10 at 14:41

... you can write allocators for the class which will maintain the pimpl's cost amortized. That adds complexity, which hits on the simplicity of the plain-vanilla copy-and-swap idiom. It's a choice. – wilhelmtell Dec 22 '10 at 14:46

4 `std::swap(this_string, that)` doesn't provide a no-throw guarantee. It provides strong exception safety, but not a no-throw guarantee. – wilhelmtell Dec 22 '10 at 14:59

7 @wilhelmtell: In C++03, there is no mention of exceptions potentially thrown by `std::string::swap` (which is called by `std::swap`). In C++0x, `std::string::swap` is `noexcept` and must not throw exceptions. – James McNellis Dec 22 '10 at 15:24

2 @wilhelmtell: I thought that was the point of swapping: it never throws and it is always O(1) (yeah, I know, `std::array` ...) – sbi Dec 22 '10 at 15:50

This answer is more like an addition and a slight modification to the answers above.

In some versions of Visual Studio (and possibly other compilers) there is a bug that is really annoying and doesn't make sense. So if you declare/define your `swap` function like this: