Exception-Safety Issues and Techniques

First, a little of the history of this topic is in order. In 1994, Tom Cargill published the seminal article "Exception Handling: A False Sense of Security" (Cargill94). It demonstrated conclusively that, at that time, the C++ community did not yet fully understand how to write exception-safe code. In fact, we didn't even know what all the important exception-safety issues were, or how to correctly reason about exception safety. Cargill challenged anyone to demonstrate a conclusive solution to this problem. Three years passed. A few people wrote partial responses to aspects of Cargill's example, but no one showed a comprehensive solution.


Then, in 1997, Guru of the Week #8 appeared on the Internet newsgroup comp.lang.c++.moderated. Number 8 generated weeks of discussion and presented the first complete solution to Cargill's challenge. Later that year, a greatly expanded version that was updated to match the latest changes to draft standard C++ and demonstrating no fewer than three complete solutions, was published in the September and November/December issues of C++ Report under the title "Exception-Safe Generic Containers." (Copies of those original articles will also appear in the forthcoming book C++ Gems II [Martin00].)

In early 1999, on the Effective C++ CD (Meyers99), Scott Meyers included a recombined version of the articles, together with Cargill's original challenge, with the updated text of his classic books Effective C++ and More Effective C++.

This miniseries has come a long way since its original publication as Guru of the Week #8. I hope you enjoy it and find it useful. Particular thanks go to fellow committee members Dave Abrahams and Greg Colvin for their insights into how to reason about exception-safety and their thoughtful critiques of several drafts of this material. Dave and Greg are, with Matt Austern, the authors of the two complete committee proposals for adding the current exception-safety guarantees into the standard library.

This miniseries tackles both major features, exception handling and templates, at once, by examining how to write exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller) generic containers. That's easy enough to say, but it's no mean feat.

So come on in, join the fun, and try your hand at implementing a simple container (a Stack that users can push and pop) and see the issues involved with making it exception-safe and exception-neutral.

Item 8. Writing Exception-Safe Code—Part 1
Exception handling and templates are two of C++’s most powerful features. Writing exception-safe code, however, can be difficult—especially in a template, when you may have no idea when (or what) a certain function might throw your way.

We’ll begin where Cargill left off—namely, by progressively creating a safe version of the Stack template he critiqued. Later on, we’ll significantly improve the Stack container by reducing the requirements on T, the contained type, and show advanced techniques for managing resources exception-safely. Along the way, we’ll find the answers to such questions as:

- What are the different "levels" of exception safety?
- Can or should generic containers be fully exception-neutral?
- Are the standard library containers exception-safe or exception-neutral?
- Does exception safety affect the design of your container's public interface?
- Should generic containers use exception specifications?

Here is the declaration of the Stack template, substantially the same as in Cargill’s article. Your mission: Make Stack exception-safe and exception-neutral. That is, Stack objects should always be in a correct and consistent state, regardless of any exceptions that might be thrown in the course of executing Stack's member functions. If any exceptions are thrown, they should be propagated seamlessly through to the caller, who can deal with them as he pleases, because he knows the context of T and we don’t.

```
template <class T> class Stack
{
public:
    Stack();
    ~Stack();
    /*...*/

private:
    T* v_;       // ptr to a memory area big
    size_t vsize_;  // enough for 'vsize_' T's
    size_t vused_;  // # of T's actually in use
};
```

Write the Stack default constructor and destructor in a way that is demonstrably exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller, without causing integrity problems in a Stack object).
Solution

Right away, we can see that Stack is going to have to manage dynamic memory resources. Clearly, one key is going to be avoiding leaks, even in the presence of exceptions thrown by T operations and standard memory allocations. For now, we'll manage these memory resources within each Stack member function. Later on in this miniseries, we'll improve on this by using a private base class to encapsulate resource ownership.

Default Construction

First, consider one possible default constructor:

```c++
// Is this safe?

template<class T>
Stack<T>::Stack()
: v_(0),
  vsize_(10),
  vused_(0)         // nothing used yet
{
  v_ = new T[vsize_]; // initial allocation
}
```

Is this constructor exception-safe and exception-neutral? To find out, consider what might throw. In short, the answer is: any function. So the first step is to analyze this code and determine which functions will actually be called, including both free functions and constructors, destructors, operators, and other member functions.

This Stack constructor first sets vsize_ to 10, then attempts to allocate some initial memory using new T[vsize_]. The latter first tries to call operator new[]() (either the default operator new[]() or one provided by T) to allocate the memory, then tries to call T::T a total of vsize_ times. There are two operations that might fail. First, the memory allocation itself, in which case operator new[]() will throw a bad_alloc exception. Second, T's default constructor, which might throw anything at all, in which case any objects that were constructed are destroyed and the allocated memory is automatically guaranteed to be deallocated via operator delete[]().

Hence the above function is fully exception-safe and exception-neutral, and we can move on to the next...what? Why is the function fully robust, you ask? All right, let's examine it in a little more detail.

1. We're exception-neutral. We don't catch anything, so if the new throws, then the exception is correctly propagated up to our caller as required.
2. **We don't leak.** If the `operator new[]()` allocation call exited by throwing a `bad_alloc` exception, then no memory was allocated to begin with, so there can't be a leak. If one of the `T` constructors threw, then any `T` objects that were fully constructed were properly destroyed and, finally, `operator delete[]()` was automatically called to release the memory. That makes us leakproof, as advertised.

I'm ignoring for now the possibility that one of the `T` destructor calls might throw during cleanup, which would call `terminate()` and simply kill the program altogether and leave events well out of your control anyway. See the point in Item 16 in which we cover information on "destructors that throw and why they're evil."

3. **We're in a consistent state whether or not any part of the `new` throws.** Now one might think that if the `new` throws, then `vsize_` has already been set to 10 when, in fact, nothing was successfully allocated. Isn't that inconsistent? Not really, because it's irrelevant. Remember if the `new` throws, we propagate the exception out of our own constructor, right? And, by definition, "exiting a constructor by means of an exception" means our `Stack` proto-object never actually got to become a completely constructed object at all. Its lifetime never started, so its state is meaningless because the object never existed. It doesn't matter what the memory that briefly held `vsize_` was set to, any more than it matters what the memory was set to after we leave an object's destructor. All that's left is raw memory, smoke, and ashes.

**Guideline**

If a function isn't going to handle (or translate or deliberately absorb) an exception, it should allow the exception to propagate up to a caller who can handle it.

**Guideline**

Always structure your code so that resources are correctly freed and data is in a consistent state even in the presence of exceptions.

All right, I'll admit it: I put the `new` in the constructor body purely to open the door for that discussion. What I'd actually prefer to write is:

```cpp
template<class T>
Stack<T>::Stack()
 : v_(new T[10]), // default allocation
   vsize_(10),
   vused_(0) // nothing used yet
{
}
```

Both versions are practically equivalent. I prefer the latter because it follows the usual good
practice of initializing members in initializer lists whenever possible.

**Destruction**

The destructor looks a lot easier, once we make a (greatly) simplifying assumption.

```cpp
template<class T>
Stack<T>::~Stack()
{
    delete[] v_;     // this can't throw
}
```

Why can't the `delete[]` call throw? Recall that this invokes `T::~T` for each object in the array, then calls `operator delete[]()` to deallocate the memory. We know that the deallocation by `operator delete[]()` may never throw, because the standard requires that its signature is always one of the following:[2]

```cpp
void operator delete[]( void* ) throw();
void operator delete[]( void*, size_t ) throw();
```

Hence, the only thing that could possibly throw is one of the `T::~T` calls, and we're arbitrarily going to have `Stack` require that `T::~T` may not throw. Why? To make a long story short, we just can't implement the `Stack` destructor with complete exception safety if `T::~T` can throw, that's why. However, requiring that `T::~T` may not throw isn't particularly onerous, because there are plenty of other reasons why destructors should never be allowed to throw at all.[3] Any class whose destructor can throw is likely to cause you all sorts of other problems sooner or later, and you can't even reliably `new[]` or `delete[]` an array of them. More on that as we continue in this miniseries.

[2] As Scott Meyers pointed out in private communication, strictly speaking this doesn't prevent someone from providing an overloaded `operator delete[]()` that does throw, but any such overload would violate this clear intent and should be considered defective.

```cpp
void operator delete[]( void* ) throw();
void operator delete[]( void*, size_t ) throw();
```

[3] Frankly, you won't go far wrong if you habitually write `throw()` after the declaration of every destructor you ever write. Even if exception specifications cause expensive checks under your current compiler, at least write all your destructors as though they were specified as `throw()`—that is, never allow exceptions to leave destructors.

**Guideline**

![Guideline](image)

*Observe the canonical exception safety rules: Never allow an exception to escape from a destructor or from an overloaded `operator delete()` or `operator delete[]()`; write every destructor and deallocation function as though it had an exception specification of "throw()"). More on this as we go on; this is an important theme.*
Item 9. Writing Exception-Safe Code—Part 2

Difficulty: 8

Now that we have the default constructor and the destructor under our belts, we might be tempted to think that all the other functions will be about the same. Well, writing exception-safe and exception-neutral copy and assignment code presents its own challenges, as we shall now see.

Consider again Cargill's Stack template:

```cpp
template <class T> class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    /*...*/
private:
    T*     v_;      // ptr to a memory area big
    size_t vsize_;  // enough for 'vsize_' T's
    size_t vused_;  // # of T's actually in use
};
```

Now write the Stack copy constructor and copy assignment operator so that both are demonstrably exception-safe (work properly in the presence of exceptions) and exception-neutral (propagate all exceptions to the caller, without causing integrity problems in a Stack object).

I love RuBoard

I love RuBoard

Solution

To implement the copy constructor and the copy assignment operator, let's use a common helper function, NewCopy, to manage allocating and growing memory. NewCopy takes a pointer to (src) and size of (srcsize) an existing T buffer, and returns a pointer to a new and possibly larger copy of the buffer, passing ownership of the new buffer to the caller. If exceptions are encountered, NewCopy correctly releases all temporary resources and propagates the exception in such a way that nothing is leaked.

```cpp
template<class T>
T* NewCopy( const T* src,
            size_t srclsize,
            size_t destsize )
```
assert( destsize >= srcsize );
T* dest = new T[destsize];
try {
    copy( src, src+srcsize, dest );
} catch(...) {
    delete[] dest; // this can't throw
    throw;         // rethrow original exception
} return dest;
}

Let's analyze this one step at a time.

1. In the new statement, the allocation might throw bad_alloc or the T::T's may throw anything. In either case, nothing is allocated and we simply allow the exception to propagate. This is both leak-free and exception-neutral.

2. Next, we assign all the existing values using T::operator=(). If any of the assignments fail, we catch the exception, free the allocated memory, and rethrow the original exception. This is again both leak-free and exception-neutral. However, there's an important subtlety here: T::operator=() must guarantee that if it does throw, then the assigned-to T object must be destructible.[4]

[4] As we progress, we'll arrive at an improved version of Stack that does not rely on T::operator=.

3. If the allocation and copy both succeed, then we return the pointer to the new buffer and relinquish ownership (that is, the caller is responsible for the buffer from here on out). The return simply copies the pointer value, which cannot throw.

Copy Construction

With NewCopy in hand, the Stack copy constructor is easy to write.

```cpp
template<class T>
Stack<T>::Stack( const Stack<T>& other )
    : v_(NewCopy( other.v_,
                  other.vsize_,
                  other.vsize_ )),
      vsize_(other.vsize_),
      vused_(other.vused_)
{ }
```

The only possible exception is from NewCopy, which manages its own resources.
Copy Assignment

Next, we tackle copy assignment.

```cpp
template<class T>
Stack<T>&
Stack<T>::operator=( const Stack<T>& other )
{
    if( this != &other )
    {
        T* v_new = NewCopy( other.v_,
                            other.vsize_,
                            other.vsize_ );
        delete[] v_;  // this can't throw
        v_ = v_new;   // take ownership
        vsize_ = other.vsize_;
        vused_ = other.vused_;
    }
    return *this;   // safe, no copy involved
}
```

Again, after the routine weak guard against self-assignment, only the `NewCopy` call might throw. If it does, we correctly propagate that exception, without affecting the `Stack` object's state. To the caller, if the assignment throws then the state is unchanged, and if the assignment doesn't throw, then the assignment and all its side effects are successful and complete.

What we see here is the following very important exception-safety idiom.

**Guideline**

> Observe the canonical exception-safety rules: In each function, take all the code that might emit an exception and do all that work safely off to the side. Only then, when you know that the real work has succeeded, should you modify the program state (and clean up) using only nonthrowing operations.

---

**Item 10. Writing Exception-Safe Code—Part 3**

**Difficulty: 9½**

Are you getting the hang of exception safety? Well, then, it must be time to throw you a curve ball. So get ready, and don't swing too soon.
Now for the final piece of Cargill's original Stack template.

```
template <class T> class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void Push(const T&);
    T Pop(); // if empty, throws exception

private:
    T* v_;      // ptr to a memory area big
    size_t vsize_;  // enough for 'vsize_' T's
    size_t vused_;  // # of T's actually in use
};
```

Write the final three Stack functions: Count(), Push(), and Pop(). Remember, be exception-safe and exception-neutral!

**Solution**

**Count()**

The easiest of all Stack's members to implement safely is Count, because all it does is copy a builtin that can never throw.

```
template<class T>
size_t Stack<T>::Count() const
{
    return vused_;  // safe, builtins don't throw
}
```

No problem.

**Push()**

On the other hand, with Push, we need to apply our now-usual duty of care.

```
template<class T>
void Stack<T>::Push( const T& t )
```
If we have no more space, we first pick a new size for the buffer and make a larger copy using \texttt{NewCopy}. Again, if \texttt{NewCopy} throws, then our own \texttt{Stack}'s state is unchanged and the exception propagates through cleanly. Deleting the original buffer and taking ownership of the new one involves only operations that are known not to throw, so the entire if block is exception-safe.

After any required grow operation, we attempt to copy the new value before incrementing our \texttt{vused\_} count. This way, if the assignment throws, the increment is not performed and our \texttt{Stack}'s state is unchanged. If the assignment succeeds, the \texttt{Stack}'s state is changed to recognize the presence of the new value, and all is well.

**Guideline**

\begin{quote}
\textbf{\texttt{Pop}() Goes the Weasel}
\end{quote}

Only one function left. That wasn't so hard, was it? Well, don't get too happy yet, because it turns out that \texttt{Pop} is the most problematic of these functions to write with complete exception safety. Our initial attempt might look something like this:

```cpp
// Hmmm... how safe is it really?

template<class T>
T Stack<T>::Pop()
{
    if( vused_ == 0)
    {
        throw "pop from empty stack";
    }
```

```cpp
}
else
{
    T result = v_[vused_-1];
    --vused;
    return result;
}

If the stack is empty, we throw an appropriate exception. Otherwise, we create a copy of the object to be returned, update our state, and return the T object. If the initial copy from v_[vused_-1] fails, the exception is propagated and the state of the Stack is unchanged, which is what we want. If the initial copy succeeds, our state is updated and the Stack is in its new consistent state, which is also what we want.

So this works, right? Well, kind of. There is a subtle flaw here that's completely outside the purview of Stack::Pop(). Consider the following client code:

```cpp
string s1(s.Pop());
string s2;
s2 = s.Pop();
```

Note that above we talked about "the initial copy" (from v_[vused_-1]). That's because there is another copy to worry about in either of the above cases—namely, the copy of the returned temporary into the destination. If that copy construction or copy assignment fails, then the Stack has completed its side effect (the top element has been popped off), but the popped value is now lost forever, because it never reached its destination (oops). This is bad news. In effect, it means that any version of Pop() that is written to return a temporary like this—and that therefore is responsible for two side effects—cannot be made completely exception-safe, because even though the function's implementation itself may look technically exception-safe, it forces clients of Stack to write exception-unsafe code. More generally, mutator functions should not return T objects by value. (See Item 19 for more about exception-safety issues for functions with multiple side effects.)

[5] For you experienced readers, yes, it's actually "zero or one copies," because the compiler is free to optimize away the second copy if the return value optimization applies. The point is that there can be a copy, so you have to be ready for it.

The bottom line—and it's significant—is this: Exception safety affects your class's design. In other words, you must design for exception safety from the outset, and exception safety is never "just an implementation detail."

**Common Mistake**

Never make exception safety an afterthought. Exception safety affects a class's design. It is never "just an implementation detail."

**The Real Problem**
One alternative—in fact, the minimum possible change[^6]—is to respecify \texttt{Pop} as follows:

[^6]: The minimum possible acceptable change, that is. You could always simply change the original version to return \texttt{T} & instead of \texttt{T} (this would be a reference to the popped \texttt{T} object, because for the time being the popped object happens to still physically exist in your internal representation), and then the caller could still write exception-safe code. But this business of returning references to "I no longer consider it there" resources is purely evil. If you change your implementation in the future, this may no longer be possible! Don't go there.

\begin{verbatim}
template<class T>
void Stack<T>::Pop( T& result )
{
  if( vused_ == 0 )
  {
    throw "pop from empty stack";
  }
  else
  {
    result = v_[vused_-1];
    --vused_;
  }
}
\end{verbatim}

This ensures that the \texttt{Stack}'s state is not changed unless the copy safely arrives in the caller's hands.

But the real problem is that, as specified, \texttt{Pop()} has two responsibilities—namely, to pop the top-most element and to return the just-popped value.

\textbf{Guideline}

\begin{itemize}
  \item Prefer cohesion. Always endeavor to give each piece of code—each module, each class, each function—a single, well-defined responsibility.
\end{itemize}

So another option (and preferable, in my opinion) is to separate the functions of "querying the top-most value" and "popping the top-most value off the stack." We do this by having one function for each.

\begin{verbatim}
template<class T>
T& Stack<T>::Top()
{
  if( vused_ == 0 )
  {
    throw "empty stack";
  }
  return v_[vused_-1];
}
\end{verbatim}
template<class T>
void Stack<T>::Pop()
{
    if (vused_ == 0)
    {
        throw "pop from empty stack";
    }
    else
    {
        --vused_;
    }
}

Incidentally, have you ever grumbled at the way the standard library containers' pop functions (for example, list::pop_back, stack::pop, etc.) don't return the popped value? Well, here's one reason to do this: It avoids weakening exception safety.

In fact, you've probably noticed that the above separated Top and Pop now match the signatures of the top and pop members of the standard library's stack<> adapter. That's no coincidence. We're actually only two public member functions away from the stack<> adapter's full public interface—namely:

template<class T>
const T & Stack<T>::Top() const
{
    if (vused_ == 0)
    {
        throw "empty stack";
    }
    else
    {
        return v_[vused_-1];
    }
}

to provide Top for const Stack objects, and:

template<class T>
bool Stack<T>::Empty() const
{
    return (vused_ == 0);
}

Of course, the standard stack<> is actually a container adapter that's implemented in terms of another container, but the public interface is the same and the rest is just an implementation detail.

There's just one more fundamental point I want to drive home. I'll leave the following with you to ponder.
"Exception-unsafe" and "poor design" go hand in hand. If a piece of code isn't exception-safe, that's generally okay and can simply be fixed. But if a piece of code cannot be made exception-safe because of its underlying design, that almost always is a signal of its poor design. Example 1: A function with two different responsibilities is difficult to make exception-safe. Example 2: A copy assignment operator that is written in such a way that it must check for self-assignment is probably not strongly exception-safe either.

You will see Example 2 demonstrated very soon in this miniseries. Note that copy assignment operators may well elect to check for self-assignment even if they don't have to—for example, they might do so for efficiency. But a copy assignment operator that has to check for self-assignment (and else would not work correctly for self-assignment) is probably not strongly exception-safe.

Item 11. Writing Exception-Safe Code—Part 4

Difficultly: 8

Mid-series interlude: What have we accomplished so far?

Now that we have implemented an exception-safe and exception-neutral Stack<T>, answer these questions as precisely as possible:

1. What are the important exception-safety guarantees?

2. For the Stack<T> that was just implemented, what are the requirements on T, the contained type?

Solution

Just as there's more than one way to skin a cat (I have a feeling I'm going to get enraged e-
mail from animal lovers), there's more than one way to write exception-safe code. In fact, there are two main alternatives we can choose from when it comes to guaranteeing exception safety. These guarantees were first set out in this form by Dave Abrahams.

1. **Basic guarantee:** *Even in the presence of exceptions thrown by* \( T \) *or other exceptions, Stack objects don't leak resources.* Note that this also implies that the container will be destructible and usable even if an exception is thrown while performing some container operation. However, if an exception is thrown, the container will be in a consistent, but not necessarily predictable, state. Containers that support the basic guarantee can work safely in some settings.

2. **Strong guarantee:** *If an operation terminates because of an exception, program state will remain unchanged.* This always implies commit-or-rollback semantics, including that no references or iterators into the container be invalidated if an operation fails. For example, if a Stack client calls Top and then attempts a Push that fails because of an exception, then the state of the Stack object must be unchanged and the reference returned from the prior call to Top must still be valid. For more information on these guarantees, see Dave Abrahams’s documentation of the SGI exception-safe standard library adaptation at: [http://www.gotw.ca/publications/xc++/da_stlsafety.htm](http://www.gotw.ca/publications/xc++/da_stlsafety.htm).

Probably the most interesting point here is that when you implement the basic guarantee, the strong guarantee often comes along for free.[7] For example, in our Stack implementation, almost everything we did was needed to satisfy just the basic guarantee—and what's presented above very nearly satisfies the strong guarantee, with little or no extra work.[8] Not half bad, considering all the trouble we went to.

[7] Note that I said "often," not "always." In the standard library, for example, vector is a well-known counter-example in which satisfying the basic guarantee does not cause the strong guarantee to come along for free.

[8] There is one subtle way in which this version of Stack still falls short of the strong guarantee. If Push() is called and has to grow its internal buffer, but then its final \( v_{[vused]} = t; \) assignment throws, the Stack is still in a consistent state, but its internal memory buffer has moved—which invalidates any previously valid references returned from Top(). This last flaw in Stack::Push() can be fixed fairly easily by moving some code and adding a try block. For a better solution, however, see the Stack presented in the second half of this miniseries. That Stack does not have the problem, and it does satisfy the strong commit-or-rollback guarantee.

In addition to these two guarantees, there is one more guarantee that certain functions must provide in order to make overall exception safety possible:

3. **Nothrow guarantee:** *The function will not emit an exception under any circumstances.* Overall exception safety isn't possible unless certain functions are guaranteed not to throw. In particular, we've seen that this is true for destructors; later in this miniseries, we'll see that it's also needed in certain helper functions, such as Swap().
Now we have some points to ponder. Note that we've been able to implement `Stack` to be not only exception-safe but fully exception-neutral, yet we've used only a single `try/catch`. As we'll see next time, using better encapsulation techniques can get rid of even this `try` block. That means we can write a strongly exception-safe and exception-neutral generic container, without using `try` or `catch`—very natty, very elegant.

For the template as we've seen it so far, `Stack` requires its instantiation type to have all of the following:

- Default constructor (to construct the `v_` buffers)
- Copy constructor (if `Pop` returns by value)
- Nonthrowing destructor (to be able to guarantee exception-safety)
- **Exception-safe** copy assignment (To set the values in `v_`, and if the copy assignment throws, then it must guarantee that the target object is still a valid `T`. Note that this is the only `T` member function that must be exception-safe in order for our `Stack` to be exception-safe.)

In the second half of this miniseries, we'll also see how to reduce even these requirements, without compromising exception safety. Along the way, we'll get an even more-detailed look at the standard operation of the statement `delete[] x;`.

**Item 12. Writing Exception-Safe Code—Part 5**

**Difficulty: 7**

All right, you’ve had enough rest—roll up your sleeves, and get ready for a wild ride.

Now we're ready to delve a little deeper into the same example, and write not just one but two new-and-improved versions of `Stack`. Not only is it, indeed, possible to write exception-safe generic containers, but by the time this miniseries is over, we'll have created no fewer than three complete solutions to the exception-safe `Stack` problem.

Along the way, we'll also discover the answers to several more interesting questions:

- How can we use more-advanced techniques to simplify the way we manage resources and get rid of the last `try/catch` into the bargain?
- How can we improve `Stack` by reducing the requirements on `T`, the contained type?
Should generic containers use exception specifications?

What do `new[]` and `delete[]` really do?

The answer to the last question may be quite different from what one might expect. Writing exception-safe containers in C++ isn't rocket science; it just requires significant care and a good understanding of how the language works. In particular, it helps to develop a habit of eyeing with mild suspicion anything that might turn out to be a function call—including user-defined operators, user-defined conversions, and silent temporary objects among the more subtle culprits—because any function call might throw.[9]

[9] Except for functions declared with an exception specification of `throw()` or certain functions in the standard library that are documented to never throw.

One way to greatly simplify an exception-safe container like `Stack` is to use better encapsulation. Specifically, we'd like to encapsulate the basic memory management work. Most of the care we had to take while writing our original exception-safe `Stack` was needed just to get the basic memory allocation right, so let's introduce a simple helper class to put all that work in one place.

```cpp
template <class T> class StackImpl
{
/*????*/:
    StackImpl(size_t size=0);
    ~StackImpl();
    void Swap(StackImpl& other) throw();

    T*     v_;      // ptr to a memory area big
    size_t vsize_;  // enough for 'vsize_' T's
    size_t vused_;  // # of T's actually in use

private:
    // private and undefined: no copying allowed
    StackImpl( const StackImpl& );
    StackImpl& operator=( const StackImpl& );
};
```

Note that `StackImpl` has all the original `Stack`'s data members so that we've essentially moved the original `Stack`'s representation entirely into `StackImpl`. `StackImpl` also has a helper function named `Swap`, which exchanges the guts of our `StackImpl` object with those of another `StackImpl`.

Your tasks:

1. Implement all three member functions of `StackImpl`, but not just any old way. Assume that at any time, the `v_` buffer must contain exactly as many constructed `T` objects as there are `T` objects in the container, no more, no less. In particular, unused space in the `v_` buffer should not contain constructed `T` objects.

2. Describe `StackImpl`'s responsibilities. Why does it exist?
3. What should /*????*/ be? How does the choice affect how StackImpl will be used? Be as specific as possible.

Solution

We won't spend much time analyzing why the following functions are fully exception-safe (work properly in the presence of exceptions) and exception-neutral (propagate all exceptions to the caller), because the reasons are pretty much the same as those we discussed in detail in the first half of this miniseries. But do take a few minutes now to analyze these solutions, and note the commentary.

Constructor

The constructor is fairly straightforward. We'll use operator new() to allocate the buffer as raw memory. (Note that if we used a new-expression like new T[size], then the buffer would be initialized to default-constructed T objects, which was explicitly disallowed in the problem statement.)

```cpp
template <class T>  
StackImpl<T>::StackImpl(size_t size)  
    : v_(static_cast<T*>(size == 0? 0: operator new(sizeof(T)*size))),  
      vsize_(size),  
      vused_(0)  
{  
}
```

Destructor

The destructor is the easiest of the three functions to implement. Again, remember what we learned about operator delete() earlier in this miniseries. (See "Some Standard Helper Functions" for full details about functions such as destroy() and swap() that appear in the next few pieces of code.)

```cpp
template <class T>  
StackImpl<T>::~StackImpl()  
{  
    destroy(v_, v_+vused_); // this can't throw  
    operator delete(v_);  
}
```

We'll see what destroy() is in a moment.
Some Standard Helper Functions

The Stack and StackImpl presented in this solution use three helper functions, one of which (swap()) also appears in the standard library: construct(), destroy(), and swap(). In simplified form, here's what these functions look like:

```cpp
// construct() constructs a new object in // a given location using an initial value //
// template <class T1, class T2>
void construct( T1* p, const T2& value )
{
    new (p) T1(value);
}
```

The above form of new is called "placement new," and instead of allocating memory for the new object, it just puts it into the memory pointed at by p. Any object new'd in this way should generally be destroyed by calling its destructor explicitly (as in the following two functions), rather than by using delete.

```cpp
// destroy() destroys an object or a range // of objects //
// template <class T>
void destroy( T* p )
{
    p->~T();
}
// template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        destroy( &*first );
        ++first;
    }
}
// swap() just exchanges two values //
// template <class T>
void swap( T& a, T& b )
{
    T temp(a); a = b; b = temp;
}
```

Of these, destroy(first, last) is the most interesting. We'll return to it a little later in the main miniseries; it illustrates more than one might think!

Swap
Finally, a simple but very important function. Believe it or not, this is the function that is instrumental in making the complete `Stack` class so elegant, especially its `operator=()`, as we'll see soon.

```
template <class T>
void StackImpl<T>::Swap(StackImpl& other) throw()
{
    swap( v_,     other.v_ );
    swap( vsize_, other.vsize_ );
    swap( vused_, other.vused_ );
}
```

To picture how `Swap()` works, say that you have two `StackImpl<T>` objects `a` and `b`, as shown in Figure 1.

**Figure 1. Two `StackImpl<T>` objects `a` and `b`**

![Diagram of two StackImpl<T> objects a and b](image)

Then executing `a.Swap(b)` changes the state to that shown in Figure 2.

**Figure 2. The same two `StackImpl<T>` objects, after `a.Swap(b)`**

![Diagram showing the state after a.Swap(b)](image)

Note that `Swap()` supports the strongest exception guarantee of all—namely, the noexcept guarantee; `Swap()` is guaranteed not to throw an exception under any circumstances. It turns out that this feature of `Swap()` is essential, a linchpin in the chain of reasoning about `Stack`'s own exception safety.
Why does StackImpl exist? Well, there's nothing magical going on here: StackImpl is responsible for simple raw memory management and final cleanup, so any class that uses it won't have to worry about those details.

**Guideline**

> Prefer cohesion. Always endeavor to give each piece of code—each module, each class, each function—a single, well-defined responsibility.

So what access specifier would you write in place of the comment "/*????*/"? Hint: The name StackImpl itself hints at some kind of "implemented-in-terms-of" relationship, and there are two main ways to write that kind of relationship in C++.

*Technique 1: Private Base Class.* The missing /*????*/ access specifier must be either protected or public. (If it were private, no one could use the class.) First, consider what happens if we make it protected.

Using protected means that StackImpl is intended to be used as a private base class. So Stack will be "implemented in terms of" StackImpl, which is what private inheritance means, and we have a clear division of responsibilities. The StackImpl base class will take care of managing the memory buffer and destroying all remaining T objects during Stack destruction, while the Stack derived class will take care of constructing all T objects within the raw memory. The raw memory management takes place pretty much entirely outside Stack itself, because, for example, the initial allocation must fully succeed before any Stack constructor body can be entered. **Item 13** begins the final phase of this miniseries, in which we'll concentrate on implementing this version.

*Technique 2: Private Member.* Next, consider what happens if StackImpl's missing /*????*/ access specifier is public.

Using public hints that StackImpl is intended to be used as a struct by some external client, because its data members are public. So again, Stack will be "implemented in terms of" StackImpl, only this time using a HAS-A containment relationship instead of private inheritance. We still have the same clear division of responsibilities. The StackImpl object will take care of managing the memory buffer and destroying all T objects remaining during Stack destruction, and the containing Stack will take care of constructing T objects within the raw memory. Because data members are initialized before a class's constructor body is entered, the raw memory management still takes place pretty much entirely outside Stack, because, for example, the initial allocation must fully succeed before any Stack constructor body can be entered.

As we'll see when we look at the code, this second technique is only slightly different from the first.
And now for an even better Stack, with fewer requirements on T—not to mention a very elegant operator=().

Imagine that the /*????*/ comment in StackImpl stood for protected. Implement all the member functions of the following version of Stack, which is to be implemented in terms of StackImpl by using StackImpl as a private base class.

```cpp
template <class T>
class Stack : private StackImpl<T> {
public:
    Stack(size_t size=0);
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void   Push(const T&);
    T&     Top();   // if empty, throws exception
    void   Pop();   // if empty, throws exception
};
```

As always, remember to make all the functions exception-safe and exception-neutral.

(Hint: There's a very elegant way to implement a fully safe operator=(). Can you spot it?)

Solution

The Default Constructor

Using the private base class method, our Stack class will look something like this (the code is shown inlined for brevity):

```cpp
template <class T>
class Stack : private StackImpl<T> {
public:
    Stack(size_t size=0) :
        StackImpl<T>(size) {
    }
};
```
Stack's default constructor simply calls the default constructor of StackImpl, that just sets the stack's state to empty and optionally performs an initial allocation. The only operation here that might throw is the new done in StackImpl's constructor, and that's unimportant when considering Stack's own exception safety. If it does happen, we won't enter the Stack constructor body and there will never have been a Stack object at all, so any initial allocation failures in the base class don't affect Stack. (See Item 8 and More Exceptional C++ Items 17 and 18, for additional comments about exiting constructors via an exception.)

Note that we slightly changed Stack's original constructor interface to allow a starting "hint" at the amount of memory to allocate. We'll make use of this in a minute when we write the Push function.

**Guideline**

![Exception Safety Icon]

*Observe the canonical exception-safety rules: Always use the "resource acquisition is initialization" idiom to isolate resource ownership and management.*

**The Destructor**

Here's the first elegance: We don't need to provide a Stack destructor. The default compiler-generated Stack destructor is fine, because it just calls the StackImpl destructor to destroy any objects that were constructed and actually free the memory. Elegant.

**The Copy Constructor**

Note that the Stack copy constructor does not call the StackImpl copy constructor. (See the previous solution for a discussion of what construct() does.)

```cpp
Stack(const Stack& other)
 : StackImpl<T>(other.vused_)
{
    while( vused_ < other.vused_ )
    {
        construct( v_+vused_, other.v_[vused_] );
        ++vused_;
    }
}
```

Copy construction is now efficient and clean. The worst that can happen here is that a T constructor could fail, in which case the StackImpl destructor will correctly destroy exactly a many objects as were successfully created and then deallocate the raw memory. One big benefit derived from StackImpl is that we could add as many more constructors as we want without putting clean-up code inside each one.

**Elegant Copy Assignment**

```cpp
Stack& operator=(const Stack& other)
{
    if( this != &other )
    {
        Stack temp(other);
        swap(temp);
    }
    return *this;
}
```
The following is an incredibly elegant and nifty way to write a completely safe copy assignment operator. It's even cooler if you've never seen the technique before.

```cpp
Stack& operator=(const Stack& other) {
    Stack temp(other); // does all the work
    Swap( temp );      // this can't throw
    return *this;
}
```

Do you get it? Take a minute to think about it before reading on.

This function is the epitome of a very important guideline that we've seen already.

**Guideline**

> Observe the canonical exception-safety rules: In each function, take all the code that might emit an exception and do all that work safely off to the side. Only then, when you know that the real work has succeeded, should you modify the program state (and clean up) using only nonthrowing operations.

It's beautifully elegant, if a little subtle. We just construct a temporary object from `other`, then call `Swap` to swap our own guts with `temp`'s, and, finally, when `temp` goes out of scope and destroys itself, it automatically cleans up our old guts in the process, leaving us with the new state.

Note that when `operator=()` is made exception-safe like this, a side effect is that it also automatically handles self-assignment (for example, `Stack s; s = s;`) correctly without further work. (Because self-assignment is exceedingly rare, I omitted the traditional `if (this != &other)` test, which has its own subtleties. See Item 38 for all the gory details.)

Note that because all the real work is done while constructing `temp`, any exceptions that might be thrown (either by memory allocation or `T` copy construction) can't affect the state of our object. Also, there won't be any memory leaks or other problems from the `temp` object, because the `Stack` copy constructor is already strongly exception-safe. Once all the work is done, we simply swap our object's internal representation with `temp`'s, which cannot throw (because `Swap` has a `throw()` exception specification, and because it does nothing but copy builtins), and we're done.

Note especially how much more elegant this is than the exception-safe copy assignment we implemented in Item 9. This version also requires much less care to ensure that it's been made properly exception-safe.

If you're one of those folks who like terse code, you can write the `operator=()` canonical form more compactly by using pass-by-value to create the temporary:
Stack& operator=(Stack temp)
{
    Swap( temp );
    return *this;
}

Stack<T>::Count()

Yes, Count() is still the easiest member function to write.

size_t Count() const
{
    return vused_;}

Stack<T>::Push()

Push() needs a little more attention. Study it for a moment before reading on.

void Push( const T& t )
{
    if( vused_ == vsize_ )    // grow if necessary
    {
        Stack temp( vsize_*2+1 );
        while( temp.Count() < vused_ )
        {
            temp.Push( v_[temp.Count()] );
        }
        temp.Push( t );
        Swap( temp );
    }
    else
    {
        construct( v_+vused_, t );
        ++vused_;     
    }
}

First, consider the simple else case: If we already have room for the new object, we attempt to construct it. If the construction succeeds, we update our vused_ count. This is safe and straightforward.

Otherwise, like last time, if we don't have enough room for the new element, we trigger a reallocation. In this case, we simply construct a temporary Stack object, push the new element onto that, and finally swap out our original guts to it to ensure they're disposed of in a tidy fashion.

But is this exception-safe? Yes. Consider:

- If the construction of temp fails, our state is unchanged and no resources have been
leaked, so that's fine.

- If any part of the loading of temp's contents (including the new object's copy construction) fails by throwing an exception, temp is properly cleaned up when its destructor is called as temp goes out of scope.

- In no case do we alter our state until all the work has already been completed successfully.

Note that this provides the strong commit-or-rollback guarantee, because the Swap() is performed only if the entire reallocate-and-push operation succeeds. Any references returned from Top(), or iterators if we later chose to provide them, would never be invalidated (by a possible internal grow operation) if the insertion is not completely successful.

**Stack<T>::Top()**

Top() hasn't changed at all.

```cpp
T& Top()
{
    if (vused_ == 0)
    {
        throw "empty stack";
    }
    return v_[vused_-1];
}
```

**Stack<T>::Pop()**

Neither has Pop(), save the new call to destroy().

```cpp
void Pop()
{
    if (vused_ == 0)
    {
        throw "pop from empty stack";
    }
    else
    {
        --vused;
        destroy( v_+vused_ );
    }
}
```

In summary, Push() has been simplified, but the biggest benefit of encapsulating the resource ownership in a separate class was seen in Stack's constructor and destructor. Thanks to StackImpl, we can go on to write as many more constructors as we like, without having to worry about clean-up code, whereas last time each constructor would have had to know about the clean-up itself.
You may also have noticed that even the lone `try/catch` we had to include in the first version of this class has now been eliminated—that is, we've written a fully exception-safe and exception-neutral generic container without writing a single `try`! (Who says writing exception-safe code is trying?)

I l@ve RuBoard
I l@ve RuBoard

**Item 14. Writing Exception-Safe Code**—Part 7

**Difficulty: 5**

*Only a slight variant—of course, `operator=( )` is still very nifty.*

Imagine that the `/*??????*/` comment in `StackImpl` stood for `public`. Implement all the member functions of the following version of `Stack`, which is to be implemented in terms of `StackImpl` by using a `StackImpl` member object.

```cpp
template <class T>
class Stack
{
  public:
    Stack(size_t size=0);
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void   Push(const T&);
    T&     Top();   // if empty, throws exception
    void   Pop();   // if empty, throws exception
  private:
    StackImpl<T> impl_;  // private implementation
};
```

Don't forget exception safety.
I l@ve RuBoard
I l@ve RuBoard

**Solution**

This implementation of `Stack` is only slightly different from the last. For example, `Count()` returns `impl_.vused_` instead of just an inherited `vused_`.

Here's the complete code:

```cpp
template <class T>
```
class Stack
{
public:
    Stack(size_t size=0)
        : impl_(size)
    {
    }

    Stack(const Stack& other)
        : impl_(other.impl_.vused_)
    {
        while( impl_.vused_ < other.impl_.vused_ )
        {
            construct( impl_.v_+impl_.vused_,
                        other.impl_.v_[impl_.vused_] );
            ++impl_.vused_;
        }
    }

    Stack& operator=(const Stack& other)
    {
        Stack temp(other);
        impl_.Swap(temp.impl_); // this can't throw
        return *this;
    }

    size_t Count() const
    {
        return impl_.vused_;
    }

    void Push( const T& t )
    {
        if( impl_.vused_ == impl_.vsize_ )
        {
            Stack temp( impl_.vsize_*2+1 );
            while( temp.Count() < impl_.vused_ )
            {
                temp.Push( impl_.v_[temp.Count()] );
            }
            temp.Push( t );
            impl_.Swap( temp.impl_ );
        }
        else
        {
            construct( impl_.v_+impl_.vused_, t );
            ++impl_.vused_;
        }
    }
}
T& Top()
{
    if( impl_.vused_ == 0 )
    {
        throw "empty stack";
    }
    return impl_.v_[impl_.vused_-1];
}

void Pop()
{
    if( impl_.vused_ == 0 )
    {
        throw "pop from empty stack";
    }
    else
    {
        --impl_.vused;
        destroy( impl_.v+impl_.vused_ );
    }
}

private:
    StackImpl<T> impl_; // private implementation
};

Whew. That's a lot of impl_'s. Which brings us to the final question in this miniseries.

I l@ve RuBoard
I l@ve RuBoard

**Item 15. Writing Exception-Safe Code**—**Part 8**

**Difficulty: 9**

*That's it—this is the final leg of the miniseries. The end of the line is a good place to stop and reflect, and that's just what we'll do for these last three problems.*

1. Which technique is better—using `StackImpl` as a private base class, or as a member object?

2. How reusable are the last two versions of `Stack`? What requirements do they put on `T`, the contained type? (In other words, what kinds of `T` can our latest `Stack` accept? The fewer the requirements are, the more reusable `Stack` will be.)

3. Should `Stack` provide exception specifications on its functions?
Solution

Let's answer the questions one at a time.

1. Which technique is better—using `StackImpl` as a private base class, or as a member object?

   Both methods give essentially the same effect and nicely separate the two concerns of memory management and object construction/destruction.

   When deciding between private inheritance and containment, my rule of thumb is always to prefer the latter and use inheritance only when absolutely necessary. Both techniques mean "is implemented in terms of," and containment forces a better separation of concerns because the using class is a normal client with access to only the used class's public interface. Use private inheritance instead of containment only when absolutely necessary, which means when:

   o You need access to the class's protected members, or

   o You need to override a virtual function, or

   o The object needs to be constructed before other base subobjects

   [10] Admittedly, in this case it's tempting to use private inheritance anyway for syntactic convenience so that we don't have to write "impl_." in so many places.

2. How reusable are the last two versions of `Stack`? What requirements do they put on `T`, the contained type?

   When writing a templated class, particularly something as potentially widely useful as a generic container, always ask yourself one crucial question: How reusable is my class? Or, to put it a different way: What constraints have I put upon users of the class, and do those constraints unduly limit what those users might want to reasonably do with my class?

   These `Stack` templates have two major differences from the first one we built. We've discussed one of the differences already. These latest `Stacks` decouple memory management from contained object construction and destruction, which is nice, but doesn't really affect users. However, there is another important difference. The new `Stacks` construct and destroy individual objects in place as needed, instead of creating default `T` objects in the entire buffer and then assigning them as needed.

   This second difference turns out to have significant benefits: better efficiency and reduced requirements on `T`, the contained type. Recall that our original `Stack` required...
T to provide four operations:

- Default constructor (to construct the v_ buffers)
- Copy constructor (if Pop returns by value)
- Nonthrowing destructor (to be able to guarantee exception safety)
- Exception-safe copy assignment (To set the values in v_, and if the copy assignmer
  throws, then it must guarantee that the target object is still a valid T. Note that this
  only T member function that must be exception-safe in order for our Stack to be
  exception-safe.)

Now, however, no default construction is needed, because the only T construction t
ever performed is copy construction. Further, no copy assignment is needed, becau
objects are never assigned within Stack or StackImpl. On the other hand, we now
need a copy constructor. This means that the new Stacks require only two things c

- Copy constructor
- Nonthrowing destructor (to be able to guarantee exception safety)

How does this measure up to our original question about usability? Well, while it's t
many classes have both default constructors and copy assignment operators, many
classes do not. (In fact, some objects simply cannot be assigned to, such as objects
contain reference members, because they cannot be reseated.) Now even these car
into Stacks, whereas in the original version they could not. That's definitely a big
advantage over the original version, and one that quite a few users are likely to app
as Stack gets reused over time.

**Guideline**

> Design with reuse in mind.

**3. Should Stack provide exception specifications on its functions?**

In short: No, because we, the authors of Stack, don't know enough, and we still probab
wouldn't want to even if we did know enough. The same is true in principle for any gener
container.

First, consider what we, as the authors of Stack, do know about T, the contained type: l
little. In particular, we don't know in advance which T operations might throw or what t
might throw. We could always get a little totalitarian about it and start dictating addition
requirements on T, which would certainly let us know more about T and maybe add som
exception specifications to Stack's member functions. However, doing that would run
completely counter to the goal of making Stack widely reusable, and so it's really out of
question.
Next, you might notice that some container operations (for example, Count()) simply return a scalar value and are known not to throw. Isn't it possible to declare these as throw()? Yes, but there are two good reasons why you probably wouldn't.

- Writing throw() limits you in the future in case you want to change the underlying implementation to a form that could throw. Loosening an exception specification always runs some risk of breaking existing clients (because the new revision of the class breaks an old promise), so your class will be inherently more resistant to change and therefore more brittle. (Writing throw() on virtual functions can also make classes less extensible, because it greatly restricts people who might want to derive from your classes. It can make sense, but such a decision requires careful thought.)

- Exception specifications can incur a performance overhead whether an exception is thrown or not, although many compilers are getting better at minimizing this. For widely-used operations and general-purpose containers, it may be better not to use exception specifications in order to avoid this overhead.

---

**Item 16. Writing Exception-Safe Code—Part 9**

**Difficulty: 8**

*How well do you understand the innocuous expression delete[] p? What are its implications when it comes to exception safety?*

And now, for the topic you've been waiting for: "Destructors That Throw and Why They're Evil."

Consider the expression `delete[] p;`, where `p` points to a valid array on the free store, which was properly allocated and initialized using `new[]`.

1. What does `delete[] p;` really do?

2. How safe is it? Be as specific as possible.

**Solution**
This brings us to a key topic, namely the innocent looking `delete[] p;`. What does it really do? And how safe is it?

**Destructors That Throw and Why They're Evil**

First, recall our standard `destroy` helper function (see the accompanying box):

```c++
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        destroy( &*first ); // calls "*first"'s destructor
        ++first;
    }
}
```

This was safe in our example above, because we required that `T` destructors never throw. But what if a contained object's destructor were allowed to throw? Consider what happens if `destroy` is passed a range of five objects. If the first destructor throws, then as it is written now, `destroy` will exit and the other four objects will never be destroyed. This is obviously not a good thing.

"Ah," you might interrupt, "but can't we clearly get around that by writing `destroy` to work properly in the face of `T`'s whose destructors are allowed to throw?" Well, that's not as clear as one might think. For example, you'd probably start writing something like this:

```c++
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        try
        {
            destroy( &*first );
        }
        catch(...)
        {
            /* what goes here? */
        }
        ++first;
    }
}
```

The tricky part is the "what goes here?" There are really only three choices: the `catch` body rethrows the exception, it converts the exception by throwing something else, or it throws nothing and continues the loop.

1. If the `catch` body rethrows the exception, then the `destroy` function nicely meets the requirement of being exception-neutral, because it does indeed allow any `T` exceptions...
to propagate out normally. However, it still doesn't meet the safety requirement that no resources be leaked if exceptions occur. Because `destroy` has no way of signaling how many objects were not successfully destroyed, those objects can never be properly destroyed, so any resources associated with them will be unavoidably leaked. Definitely not good.

2. If the `catch` body converts the exception by throwing something else, we've clearly failed to meet both the neutrality and the safety requirements. Enough said.

3. If the `catch` body does not throw or rethrow anything, then the `destroy` function nicely meets the safety requirement that no resources be leaked if an exception is thrown.[11]

   [11] True, if a `T` destructor could throw in a way that its resources might not be completely released, then there could still be a leak. However, this isn't `destroy`'s problem...this just means that `T` itself is not exception-safe. But `destroy` is still properly leak-free in that it doesn't fail to release any resources that it should (namely the `T` objects themselves).

However, it obviously fails to meet the neutrality requirement that `T` exceptions be allowed to pass through because exceptions are absorbed and ignored (as far as the caller is concerned, even if the `catch` body does attempt to do some sort of logging).

I've seen people suggest that the function should catch the exception and "save" it while continuing to destroy everything else, then rethrow it at the end. That too isn't a solution—for example, it can't correctly deal with multiple exceptions should multiple `T` destructors throw (even if you save them all until the end, you can end by throwing only one of them and the others are silently absorbed). You might be thinking of other alternatives, but trust me, they all boil down to writing code like this somewhere, because you have a set of objects and they all need to be destroyed. Someone, somewhere, is going to end up writing exception-unsafe code (at best) if `T` destructors are ever allowed to throw.

Which brings us to the innocent-looking `new[]` and `delete[]`.

The issue with both of these is that they have fundamentally the same problem we just described for `destroy`. For example, consider the following code:

```cpp
T* p = new T[10];
delete[] p;
```

Looks like normal, harmless C++, doesn't it? But have you ever wondered what `new[]` and `delete[]` do if a `T` destructor throws? Even if you have wondered, you can't know the answer for the simple reason that there is none. The standard says you get undefined behavior if a `T` destructor throws anywhere in this code, which means that any code that allocates or deallocates an array of objects whose destructors could throw can result in undefined behavior. This may raise some eyebrows, so let's see why this is so.

First, consider what happens if the constructions all succeed and, then, during the `delete[]` operation, the fifth `T` destructor throws. Then `delete[]`, has the same catch-22 problem[12].
outlined above for **destroy**. It can't allow the exception to propagate because then the remaining \( T \) objects would be irretrievably undestroyable, but it also can't translate or absorb the exception because then it wouldn't be exception-neutral.


Second, consider what happens if the fifth constructor throws. Then the fourth object's destructor is invoked, then the third's, and so on until all the \( T \) objects that were successfully constructed have again been destroyed, and the memory is safely deallocated. But what if things don't go so smoothly? In particular, what if, after the fifth constructor throws, the fourth object's destructor throws? And, if that's ignored, the third's? You can see where this is going.

If destructors may throw, then neither `new[]` nor `delete[]` can be made exception-safe and exception-neutral.

The bottom line is simply this: *Don't ever write destructors that can allow an exception to escape.*[13] If you do write a class with such a destructor, you will not be able to safely even `new[]` or `delete[]` an array of those objects. All destructors should always be implemented as though they had an exception specification of `throw()`—that is, no exceptions must ever be allowed to propagate.

[13] As of the London meeting in July of 1997, the draft makes the blanket statement: "No destructor operation defined in the C++ Standard Library will throw an exception." Not only do all the standard classes have this property, but in particular it is not permitted to instantiate a standard container with a type whose destructor does throw. The rest of the guarantees I'm going to outline were fleshed out at the following meeting (Morristown, N.J., November 1997, which was the meeting at which the completed standard was voted out).

**Guideline**

- **Observe the canonical exception safety rules**: Never allow an exception to escape from a destructor or from an overloaded `operator delete()` or `operator delete[]()`. Write every destructor and deallocation function as though it had an exception specification of "throw()."

Granted, some may feel that this state of affairs is a little unfortunate, because one of the original reasons for having exceptions was to allow both constructors and destructors to report failures (because they have no return values). This isn't quite true, because the intent was mainly for constructor failures (after all, destructors are supposed to destroy, so the scope for failure is definitely less). The good news is that exceptions are still perfectly useful for reporting construction failures, including array and array-`new[]` construction failures, because there they can work predictably, even if a construction does throw.

**Safe Exceptions**

The advice "be aware, drive with care" certainly applies to writing exception-safe code for...
containers and other objects. To do it successfully, you do have to meet a sometimes significant extra duty of care. But don't get unduly frightened by exceptions. Apply the guidelines outlined above—that is, isolate your resource management, use the "update a temporary and swap" idiom, and never write classes whose destructors can allow exceptions to escape—and you'll be well on your way to safe and happy production code that is both exception-safe and exception-neutral. The advantages can be both concrete and well worth the trouble for your library and your library's users.

For your convenience (and, hopefully, your future review), here is the "exception safety canonical form" summarized in one place.

**Guideline**

- Observe the canonical exception-safety rules: (1) Never allow an exception to escape from a destructor or from an overloaded `operator delete()` or `operator delete[]()`; write every destructor and deallocation function as though it had an exception specification of "throw()." (2) Always use the "resource acquisition is initialization" idiom to isolate resource ownership and management. (3) In each function, take all the code that might emit an exception and do all that work safely off to the side. Only then, when you know that the real work has succeeded, should you modify the program state (and clean up) using only nonthrowing operations.

---

**Item 17. Writing Exception-Safe Code—Part 10**

**Difficulty: 9½**

The end—at last. Thank you for considering this miniseries. I hope you've enjoyed it.

At this point, you're probably feeling a little drained and more than a little tired. That's understandable. So here's a final question as a parting gift—it's designed to make everyone remember the equally (if not more) tired people who had to figure this stuff out on their own from first principles and then scrambled hard to get reasonable exception-safety guarantees put into the standard library at the last minute. It's appropriate at this time to repeat public thanks to Dave Abrahams, Greg Colvin, Matt Austern, and all the other "exceptional" people who helped get the current safety guarantees into the standard library—and who managed to complete the job literally days before the standard was frozen in November 1997, at the ISO WG21 / ANSI J16 meeting at Morristown, N.J., USA.

Is the C++ standard library exception-safe?

Explain.
Solution

Exception Safety and the Standard Library

Are the standard library containers exception-safe and exception-neutral? The short answer is: Yes.\[14\]

[14] Here, I'm focusing my attention on the containers and iterators portion of the standard library. Other parts of the library, such as iostreams and facets, are specified to provide at least the basic exception-safety guarantee.

- All iterators returned from standard containers are exception-safe and can be copied without throwing an exception.

- All standard containers must implement the basic guarantee for all operations: They are always destructible, and they are always in a consistent (if not predictable) state even in the presence of exceptions.

- To make this possible, certain important functions are required to implement the nothrow guarantee (are required not to throw)—including `swap` (the importance of which was illustrated by the example in the previous Item), `allocator<T>::deallocate` (the importance of which was illustrated by the discussion of `operator delete()` at the beginning of this miniseries) and certain operations of the template parameter types themselves (especially, the destructor, the importance of which was illustrated in Item 16 by the discussion headed "Destructors That Throw and Why They're Evil").

- All standard containers must also implement the strong guarantee for all operations (with two exceptions). They always have commit-or-rollback semantics so that an operation such as an `insert` either succeeds completely or else does not change the program state at all. "No change" also means that failed operations do not affect the validity of any iterators that happened to be already pointing into the container.

- There are only two exceptions to this point. First, for all containers, multi-element inserts ("iterator range" inserts) are never strongly exception-safe. Second, for `vector<T>` and `deque<T>` only, inserts and erases (whether single- or multi-element) are strongly exception-safe as long as `T`'s copy constructor and assignment operator do not throw. Note the consequences of these particular limitations. Unfortunately, among other things, this means that inserting into and erasing from a `vector<string>` or a `vector<vector<int>>`, for example, are not strongly exception-safe.

- Why these particular limitations? Because to roll back either kind of operation isn't possible without extra space/time overhead, and the standard did not want to require that overhead in the name of exception safety. All other container operations can be
made strongly exception-safe without overhead. So if you ever insert a range of elements into a container, or if T's copy constructor or assignment operator can throw and you insert into or erase from a `vector<T>` or a `deque<T>`, the container will not necessarily have predictable contents afterward and iterators into it may have been invalidated.

What does this mean for you? Well, if you write a class that has a container member and you perform range insertions, or you write a class that has a member of type `vector<T>` or `deque<T>`, and T's copy constructor or assignment operator can throw, then you are responsible for doing the extra work to ensure that your own class's state is predictable if exceptions do occur. Fortunately, this "extra work" is pretty simple. Whenever you want to insert into or erase from the container, first take a copy of the container, then perform the change on the copy. Finally, use `swap` to switch over to using that new version after you know that the copy-and-change steps have succeeded.

**Item 18. Code Complexity—Part 1**

**Difficulty: 9**

This problem presents an interesting challenge: How many execution paths can there be in a simple three-line function? The answer will almost certainly surprise you.

How many execution paths could there be in the following code?

```cpp
String EvaluateSalaryAndReturnName( Employee e )
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

To provide a little structure here, you should start by relying on the following three assumptions, and then try to expand on them.

1. Different orders of evaluating function parameters are ignored, and failed destructors are ignored.

2. Called functions are considered atomic.

3. To count as a different execution path, an execution path must be made up of a unique sequence of function calls performed and exited in the same way.
Solution

First, let's think about the implications of the given assumptions:

1. Different orders of evaluating function parameters are ignored, and exceptions thrown by destructors are ignored. Follow-up question for the intrepid: How many more execution paths are there if destructors are allowed to throw?

   
   [15] Never allow an exception to propagate from a destructor. Code that does this can't be made to work well. See Item 16 for more about "destructors that throw and why they're evil."

2. Called functions are considered atomic. For example, the call "e.Title()" could throw for several reasons (for example, it could throw an exception itself, it could fail to catch an exception thrown by another function it has called, or it could return by value and the temporary object's constructor could throw). All that matters to the function is whether performing the operation e.Title() results in an exception being thrown or not.

3. To count as a different execution path, an execution path must be made up of a unique sequence of function calls performed and exited in the same way.

So, how many possible execution paths are there? Answer: 23 (in just three lines of code!).

<table>
<thead>
<tr>
<th>If you found:</th>
<th>Rate yourself:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Average</td>
</tr>
<tr>
<td>4–14</td>
<td>Exception-aware</td>
</tr>
<tr>
<td>15–23</td>
<td>Guru material</td>
</tr>
</tbody>
</table>

The 23 are made up of:

- 3 nonexceptional code paths
- 20 invisible exceptional code paths

By nonexceptional code paths I mean paths of execution that happen even if there are no exceptions thrown. Nonexceptional code paths result from normal C++ program flow. On the other hand, by exceptional code paths I mean those paths of execution that arise as a result of an exception being thrown or propagated, and I'll consider those paths separately.

Nonexceptional Code Paths

For the nonexceptional execution paths, the trick was to know C/C++'s short-circuit
evaluation rule.

```cpp
if( e.Title() == "CEO" || e.Salary() > 100000 )
```

1. If `e.Title() == "CEO"` evaluates to `true`, then the second part of the condition doesn't need to be evaluated (for example, `e.Salary()` will never be called), but the `cout` will be performed.

With suitable overloads for `==`, `||`, and/or `>` in the `if`'s condition, the `||` could actually turn out to be a function call. If it is a function call, the short-circuit evaluation rule would be suppressed and both parts of the condition would be evaluated all the time.

2. If `e.Title() != "CEO"` but `e.Salary() > 100000`, both parts of the condition will be evaluated and the `cout` will be performed.

3. If `e.Title() != "CEO"` and `e.Salary() <= 100000`, the `cout` will not be performed.

### Exceptional Code Paths

This leaves the exceptional execution paths.

```cpp
String EvaluateSalaryAndReturnName( Employee e )
```

4. The argument is passed by value, which invokes the `Employee` copy constructor. This copy operation might throw an exception.

   * String's copy constructor might throw while copying the temporary return value into the caller's area. We'll ignore this one, however, because it happens outside this function (and it turns out that we have enough execution paths of our own to keep us busy anyway).

   ```cpp
   if( e.Title() == "CEO" || e.Salary() > 100000 )
   ```

5. The `Title()` member function might itself throw, or it might return an object of class type by value, and that copy operation might throw.

6. To match a valid `operator==()`, the string literal may need to be converted to a temporary object of class type (probably the same as `e.Title()`'s return type), and that construction of the temporary might throw.

7. If `operator==()` is a programmer-supplied function, it might throw.

8. Similar to #5, `Salary()` might itself throw, or it might return a temporary object and this construction operation might throw.

9. Similar to #6, a temporary object may need to be constructed and this construction might throw.
10. Similar to #7, this might be a programmer-provided function and therefore might throw.

11. Similar to #7 and #10, this might be a programmer-provided function and therefore might throw.

```cpp
cout << e.First() << " " << e.Last() << " is overpaid" << endl;
```

12. As documented in the C++ Standard, any of the five calls to `operator<<` might throw.

13. Similar to #5, `First()` and/or `Last()` might throw, or each might return a temporary object and those construction operations might throw.

```cpp
return e.First() + " " + e.Last();
```

14. Similar to #5, `First()` and/or `Last()` might throw, or each might return a temporary object and those construction operations might throw.

15. Similar to #6, a temporary object may need to be constructed and this construction might throw.

16. Similar to #7, this might be a programmer-provided function and therefore might throw.

**Guideline**

Always be exception-aware. Know what code might emit exceptions.

One purpose of this Item was to demonstrate just how many invisible execution paths can exist in simple code in a language that allows exceptions. Does this invisible complexity affect the function's reliability and testability? See the following Item for the answer.

---

**Item 19. Code Complexity—Part 2**

**Difficulty: 7**

The challenge: Take the three-line function from Item 18 and make it exception-safe. This exercise illustrates some important lessons about exception safety.
Is the function from Item 18 exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller)?

```c++
String EvaluateSalaryAndReturnName( Employee e )
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

Explain your answer. If it is exception-safe, does it support the basic guarantee, the strong guarantee, or the nothrow guarantee? If not, how must it be changed to support one of these guarantees?

Assume that all called functions are strongly exception-safe (might throw but do not have side effects if they do throw), and that any objects being used, including temporaries, are exception-safe (clean up their resources when destroyed).

To recap the basic, strong, and nothrow guarantees, see Item 11. In brief, the basic guarantee ensures destructibility and no leaks; the strong guarantee, in addition, ensures full commit-or-rollback semantics; and the nothrow guarantee ensures that a function will not emit an exception.

Solution

First, before we get into the solution proper, a word about assumptions.

As the problem stated, we will assume that all called functions—including the stream functions—are strongly exception-safe (might throw but do not have side effects), and that any objects being used, including temporaries, are exception-safe (clean up their resources when destroyed).

Streams happen to throw a monkey wrench into this because of their possible "un-rollbackable" side effects. For example, `operator<<` might throw after emitting part of a string that can't be "un-emitted"; also, the stream's error state may be set. We will ignore those issues for the most part; the point of this discussion is to examine how to make a function exception-safe when the function is specified to have two distinct side effects.

So here's the question again: Is the function from Item 18 exception-safe (works properly in the presence of exceptions) and exception-neutral (propagates all exceptions to the caller)?
String EvaluateSalaryAndReturnName( Employee e )
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}

As written, this function satisfies the basic guarantee: In the presence of exceptions, the function does not leak resources.

This function does not satisfy the strong guarantee. The strong guarantee says that if the function fails due to an exception, program state must not be changed. EvaluateSalaryAndReturnName(), however, has two distinct side effects (as hinted at in the function's name).

- An "...overpaid..." message is emitted to cout.
- A name string is returned.

As far as the second side effect is concerned, the function already meets the strong guarantee because if an exception occurs the value will never be returned. As far as the first side effect is concerned, though, the function is not exception-safe for two reasons:

- If an exception is thrown after the first part of the message has been emitted to cout before the message has been completed (for example, if the fourth operator<< throws), then a partial message was emitted to cout.\[16\]

\[16\] If you’re thinking that it’s a little pedantic to worry about whether a message is completely cout’ed or not, you’re partly right. In this case, maybe no one would care. However, the same principle applies to any function that attempts to perform two side effects, and that’s why the following discussion is useful.

- If the message was emitted successfully but an exception occurs later in the function (for example, during the assembly of the return value), then a message was emitted to cout even though the function failed because of an exception.

Finally, the function clearly does not satisfy the nothrow guarantee: Lots of operations might throw, and there's no try/catch block or throw() specification in sight.

**Guideline**

Understand the basic, strong, and nothrow exception-safety guarantees.

To meet the strong guarantee, either both effects are completed or an exception is thrown and neither effect is performed.
Can we accomplish this? Here’s one way we might try it:

```c++
// Attempt #1: An improvement?
//
String EvaluateSalaryAndReturnName( Employee e )
{
    String result = e.First() + " " + e.Last();
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
    return result;
}
```

This isn't bad. Note that we've replaced the `endl` with a newline character (which isn't exactly equivalent) in order to get the entire string into one `operator<<` call. (Of course, this doesn't guarantee that the underlying stream system won't itself fail partway through writing the message, resulting in incomplete output, but at least we've done the best we can do at this high level.)

We still have one minor quibble, however, as illustrated by the following client code:

```c++
// A problem...
//
String theName;
theName = EvaluateSalaryAndReturnName( bob );
```

The `String` copy constructor is invoked because the result is returned by value, and the copy assignment operator is invoked to copy the result into `theName`. If either copy fails, then the function has completed its side effects (since the message was completely emitted and the return value was completely constructed), but the result has been irretrievably lost (oops).

Can we do better, and perhaps avoid the problem by avoiding the copy? For example, we could let the function take a non-`const String` reference parameter and place the return value in that.

```c++
// Attempt #2: Better now?
//
void EvaluateSalaryAndReturnName( Employee e, String&  r )
{
    String result = e.First() + " " + e.Last();
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
}
This may look better, but it isn't, because the assignment to \( r \) might still fail, which leaves us with one side effect complete and the other incomplete. Bottom line, this attempt doesn't really buy us much.

One way to solve the problem is to return a pointer to a dynamically allocated \texttt{String}. But the best solution is to go a step farther and return the pointer in an \texttt{auto_ptr}.

\[
\text{EvaluateSalaryAndReturnName( Employee e )} \\
\{
    \texttt{auto\_ptr\textless String\r
\rangle result} \\
    = \texttt{new String( e.First() + " " + e.Last() )}; \\
    \textit{if( e.Title() == "CEO" || e.Salary() > 100000 )} \\
    \{
        \texttt{String message = (*result) + " is overpaid\n";} \\
        \texttt{cout \ll message;} \\
    \}
\]

This does the trick, because we have effectively hidden all the work to construct the second side effect (the return value), while we ensured that it can be safely returned to the caller using only nonthrowing operations after the first side effect has completed (the printing of the message). We know that once the function is complete, the returned value will make it successfully into the hands of the caller and be correctly cleaned up in all cases. If the caller accepts the returned value, the act of accepting a copy of the \texttt{auto\_ptr} causes the caller to take ownership; and if the caller does not accept the returned value, say by ignoring the return value, the allocated \texttt{String} will automatically be cleaned up as the temporary \texttt{auto\_ptr} holding it is destroyed. The price for this extra safety? As often happens when implementing strong exception safety, the strong safety comes at the (usually minor) cost of some efficiency—here, the extra dynamic memory allocation. But, when it comes to trading off efficiency for predictability and correctness, we ought to prefer the latter two.

Let's discuss, for a moment, exception safety and multiple side effects. In this case, it turned out to be possible in Attempt \#3 to perform both side effects with essentially commit-or-rollback semantics (except for the stream issues). It was possible because there turned out to be a technique by which the two effects could be performed atomically—that is, all the "real" preparatory work for both could be completed in such a way that actually performing the visible side effects could be done using only nonthrowing operations.

Even though we were lucky this time, it's not always that simple. It's impossible to write
strongly exception-safe functions that have two or more unrelated side effects that cannot be performed atomically (for example, what if the two side effects here had been to emit one message to `cout` and another to `cerr`?), since the strong guarantee states that in the presence of exceptions "program state will remain unchanged"—in other words, if there's an exception, there must be no side effects. When you come across a case in which the two side effects cannot be made to work atomically, usually the only way to get strong exception safety is to split the one function into two others that can be performed atomically. That way, at least, the fact that they can't be done atomically is visible to the calling code.

**Guideline**

*Prefer cohesion. Always endeavor to give each piece of code—each module, each class, each function—a single, well-defined responsibility.*

In summary, this Item illustrates three important points.

1. Providing the strong exception-safety guarantee often (but not always) requires you to trade off performance.

2. If a function has multiple unrelated side effects, it cannot always be made strongly exception-safe. If not, it can be done only by splitting the function into several functions, each of whose side effects can be performed atomically.

3. Not all functions need to be strongly exception-safe. Both the original code and Attempt #1 satisfy the basic guarantee. For many clients, Attempt #1 is sufficient and minimizes the opportunity for side effects to occur in the exceptional situations, without requiring the performance tradeoffs of Attempt #3.

There's a postscript to this solution regarding streams and side effects.

In this Item's problem statement, I said in part: Assume that all called functions are exception-safe (might throw but do not have side effects if they do throw), and that any objects being used, including temporaries, are exception-safe (clean up their resources when destroyed).

As it turns out, our assumption that no called function has side effects cannot be entirely true. In particular, there is no way to guarantee that the stream operations will not fail after partly emitting a result. This means that we can't get true commit-or-rollback fidelity from any function that performs stream output, at least not on these standard streams.

Another issue is that if the stream output fails, the stream state will have changed. We currently do not check for that or recover from it, but it is possible to further refine the function to catch stream exceptions and reset `cout`'s error flags before rethrowing the exception to the caller.
I l@ve RuBoard