**This is an updated version of an article that appeared in the September and November-December 1997 issues of the *C++ Report*.**

# Exception-Safe Generic Containers ¤

by Herb Sutter ¤

Exception handling and generic programming are two of C++'s most powerful features. Both, however, require exceptional care, and writing an efficient reusable generic container is nearly as difficult as writing exception-safe code. ¤

This article tackles both of these major features 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. If you have any doubts on that score, see Tom Cargill's excellent article, Exception Handling: A False Sense of Security. ¤

This article begins where Cargill's left off, namely by presenting an exception-neutral version of the `Stack` template he critiques. In the end, 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 questions like the following: ¤

- 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 -neutral? ¤
- Does exception safety affect the design of your container's public interface? ¤
- Should generic containers use exception specifications? ¤

## The `Stack<>` Container ¤

Here is the declaration of the `Stack` template, substantially the same as in Cargill's article. Our mission: to make `Stack` 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, and 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();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Size() 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
};
```

Before reading on, stop and think about this container and consider: What are the exception-safety issues? How can this class be made exception-neutral, so that any exceptions are propagated to the caller without causing integrity problems in a Stack object? ¤

## Default Construction ¤

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, we'll improve on this by using a private base class to encapsulate resource ownership. ¤

First, consider one possible default constructor: ¤

```
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? 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 what 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; and second, T's default constructor, which might throw anything at all, and 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 we can move on to the next ... ... what? Why is it exception-safe, 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 leak-proof, as advertised. (I'm ignoring for now the possibility that one of the T destructor calls might throw during the cleanup, which would call terminate() and simply kill the program altogether and leave events well out of your control anyway. See below for more information on Destructors That Throw and Why They're Evil.) ¤

3. *We're in a consistent state whether any part of the `new` throws or not*. Now, you 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, and hence 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. ¤

All right, I'll admit it... I put the `new` in the constructor body purely to open the door for that last #3 discussion. What I'd actually prefer to write is: ¤

```
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:

```
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. Now, we know that the deallocation by `operator delete[]` may never throw, because its signature is always one of the following: ¤

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

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. 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. (Frankly, you won't go far wrong if you just 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.) Any class whose destructor can throw is likely to cause you all sorts of other problems anyway sooner or later, and you can't even reliably `new[]` or `delete[]` an array of them. More on that later. ¤

## Copy Construction and Copy Assignment ¤

The next few functions will 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. ¤

```
template<class T>
T* NewCopy( const T* src,
    size_t   srcsize,
    size_t   destsize ) {
  assert( destsize >= srcsize );
  T* dest = new T[destsize];
  try {
    copy( src, src+srcsize, dest ); // copy is part of the STL;
  } 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 leak-free and exception-neutral. ¤
2. Next, we assign all the existing values using copy, and copy invokes 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 unchanged. (Later, I will show an improved version of Stack which does not rely on T::operator=.) ¤
3. If the allocation and copy both succeeded, 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. ¤

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

```
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. Next, we tackle copy assignment: ¤

```
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 of its side effects are successful and complete. ¤

## `Size()`, `Push()`, and `Pop()` ¤

The easiest of all `Stack`'s members to implement safely is `Size`, because all it does is copy a built-in which can never throw: ¤

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

However, with `Push` we need to apply our now-usual duty of care: ¤

```
template<class T>
void Stack<T>::Push( const T& t ) {
  if( vused_ == vsize_ )               // grow if necessary
  {                                    // by some grow factor
    size_t vsize_new = vsize_*2+1;
    T* v_new = NewCopy( v_, vsize_, vsize_new );
    delete[] v_;                       // this can't throw
    v_ = v_new;                        // take ownership
    vsize_ = vsize_new;
  }
  v_[vused_] = t;
  ++vused_;
}
```

If we have no more space, we first pick a new size for the buffer and make a larger copy using `NewCopy`. Again, if `NewCopy` throws then our own `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 `vused_` count. This way, if the assignment throws, the increment is not performed and our `Stack`'s state is unchanged. If the assignment succeeds, the `Stack`'s state is changed to recognize the presence of the new value, and all is well.

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

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

```
int i(s.Pop());
int j;
j = 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. (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.) 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 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. ¤

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." One alternative is to respecify `Pop` as follows: ¤

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

A potentially tempting alternative is to simply change the original version to return `T&` instead of `T` (this would be a reference to the popped `T` object, since 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 just purely evil. If you change your implementation in the future, this may no longer be possible! Don't go there. ¤

The modified `Pop` ensures that the `Stack`'s state is not changed unless the copy safely arrives in the caller's hands. Another option (and preferable, in my opinion) is to separate the functions of "querying the topmost value" and "popping the topmost value off the stack." We do this by having one function for each: ¤

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

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 (e.g., `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. ¤

## Levels of Safety: The Basic and Strong Guarantees ¤

Just as there's more than one way to skin a cat (somehow I have a feeling I'm going to get enraged email 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 `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. (This is similar to every `Stack` member function leaving the object in what Jack Reeves terms a good — but never a bad or an undefined — state. For details, consult Reeves' article, Coping with Exceptions.) ¤

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` which 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' documentation of the ◦SGI exception-safe standard library adaptation. ¤

Probably the most interesting point here is that when you implement the basic guarantee, the strong guarantee often comes along for free. (Note that I said "often," not "always." In the standard library, for example, `vector` is a well-known counterexample where satisfying the basic guarantee does not cause the strong guarantee to come along for free.) 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. Not half bad, considering all the trouble we went to. ¤

(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 and all, 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 below — that `Stack` does not have this problem, and it does satisfy the strong commit-or-rollback guarantee.) ¤

## 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 below, using better encapsulation techniques can get rid of even this try block. That means we can write a fully exception-safe and exception-neutral generic container without using try or catch... which really is pretty cool. ¤

As originally defined, `Stack` requires its instantiation type to have a: ¤

- 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 unchanged; note that this is the only `T` member function which must be exception-safe in order for our `Stack` to be exception-safe) ¤

Next, we'll see how to reduce even these requirements without compromising exception safety, and along the way we'll get an even more detailed look at the standard operation of the statement `delete[] x;`. ¤

## Delving Deeper ¤

Now I'll delve a little deeper into the `Stack` example, and write not just one but two new-and-improved versions of the template. Not only is it possible to write exception-safe generic containers, but between the last approach and this one I'll have demonstrated no less than three different complete solutions to the exception-safe `Stack` problem. ¤

Along the way, I'll also answer 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 may be quite different than you 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 (except for functions declared with an exception specification of `throw()`, or certain functions in the standard library that are documented to never throw). ¤

## An Improved `Stack` ¤

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 of that work in one place: ¤

```
template <class T> class StackImpl {
/*????*/:
  StackImpl(size_t size=0)
  : v_( static_cast<T*>
    ( size == 0
    ? 0
    : ::operator new(sizeof(T)*size) ) ),
    vsize_(size),
    vused_(0)
  { }
  ~StackImpl() {
    destroy( v_, v_+vused_ );          // this can't throw
    ::operator delete( v_ );
  }
  void Swap(StackImpl& other) throw() {
    swap(v_, other.v_);
    swap(vsize_, other.vsize_);
    swap(vused_, other.vused_);
  }
  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
};
```

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. We won't spend much time analyzing why this class is fully exception-safe (works properly in the presence of exceptions) and exception- neutral (propagates all exceptions to the caller), because the reasons are pretty much the same as those we dissected in detail above. ¤

Note that StackImpl has all of 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. ¤

Before reading on, stop and think about this class and consider: What access specifier would you write in place of the comment "/*????*/"? And how exactly might you use a class like this to simplify Stack? (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 can even be called. So far, so good. ¤

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

```
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, which just sets the stack's state to empty and optionally performs an initial allocation. The only operation here which 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 and there will never have been a Stack object at all, so any initial allocation failures in the base class don't affect Stack.

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. ¤

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. ¤

```
Stack(const Stack& other)
  : StackImpl<T>(other.vused_)
{
  while( vused_ < other.vused_ ) {
  construct( v_+vused_,                 // see Sidebar 1 for
         other.v_[vused_] );            // info on construct
  ++vused_;
```

```
    }
  }
```

Copy construction now becomes efficient and elegant. The worst that can happen here is that a `T` constructor could fail, in which case the `StackImpl` destructor will correctly destroy exactly as 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 cleanup code in each one.

```
  Stack& operator=(const Stack& other) {
    Stack temp(other);       // does all the work
    Swap( temp );            // this can't throw — see Sidebar 1
                             // for info on swap
    return *this;
  }
```

Copy assignment is even more elegant, if a little subtle: we 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. Also, when `operator=` is made exception-safe like this, a side effect is that it usually also automatically handles self-assignment (e.g., `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 subtle problems. See ◦Guru of the Week #11 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 fully exception-neutral. 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 built-ins), and we're done. ¤

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

```
  size_t Count() const {
    return vused_;
  }
```

Yes, `Count` is still the easiest member function to write. ¤

```
  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: ¤

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

2. 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. ¤

3. 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. If we were supporting iterators into this container, for instance, they would never be invalidated (by a possible internal grow operation) if the insertion is not completely successful.

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

The `Top` function hasn't changed at all. ¤

```
void Pop() {
  if( vused_ == 0) {
    throw "pop from empty stack";
  } else {
    --vused_;
    destroy( v_+vused_ );          // see Sidebar 1 for info
  }                                // on destroy
 }
}
```

Neither has `Pop`, save the new call to destroy. ¤

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 cleanup code, whereas last time each constructor would have had to know about the cleanup 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?) ¤

## 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 subobjects are created 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. ¤

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

## Which Technique Is Better? ¤

So, how do you choose between using `StackImpl` as a private base class or as a member object? After all, both 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 to always 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' public interface. Use private inheritance instead of containment only when absolutely necessary, which means when either: ¤

- you need access to the class's protected members; ¤
- you need to override a virtual function; or ¤
- the object needs to be constructed before other base subobjects. ¤

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

(In the special case of an empty class, private inheritance may allow for the generation of more compact objects. See Scott Meyers' article, Counting Objects in C++, for details.)

## Relaxing the Requirements on `T` ¤

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? That is, 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 one we originally considered. We've discussed one already: They 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. Our original `Stack`s from last time 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 assignment throws then it must guarantee that the target object is unchanged; note that this is the only `T` member function which 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 that's ever performed is copy construction. Further, no copy assignment is needed because `T` objects are never assigned within `Stack` or `StackImpl`. On the other hand, we now always need a copy constructor. This means that the new `Stacks` require only two things of `T`: ¤

- 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 true that many classes have both default constructors and copy assignment operators, many useful classes do not (In fact, some objects simply cannot be assigned to, such as objects that contain reference members because these cannot be reseated.) Now even these can be put 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 appreciate as `Stack` gets reused over time. ¤

## Should `Stack` Provide Exception Specifications? ¤

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

First, consider what we as the authors of `Stack` do know about `T`, the contained type: The answer is, precious little. In particular, we don't know in advance which `T` operations might throw or what they might throw. We could always get a little fascist about it and start dictating additional requirements on `T`, which would certainly let us know more about `T` and maybe add some useful 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 the question. ¤

Next, you might notice that some container operations (e.g., `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: ¤

1. Writing `throw()` limits you in the future in case you want to change the underlying implementation to a form which 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), and 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.) ¤ Sutter, P132

2. 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. ¤

## Destructors That Throw and Why They're Evil ¤

This brings us to our last topic, namely the innocent-looking `delete[] p;`. What does it really do? And how safe is it? ¤

First, recall our standard `destroy` helper function (see Sidebar 1): ¤

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last ) {
  while( first != last ) {
    destroy( first );         // calls "*first"'s dtor
    ++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? Well, 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 you might think. For example, you'd probably start writing something like this: ¤

```
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: either the catch body rethrows the exception, or 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 and 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. (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, obviously it 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 only end by throwing 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 non-exception-neutral 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: ¤

```
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 (pun intended) 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.* (The C++ standard makes the blanket statement: "No destructor operation defined in the C++ Standard Library will throw an exception." Not only do all of the standard classes have this property, but in particular it is illegal to instantiate a standard container with a type whose destructor does throw.) 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. ¤

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. ¤

## Acknowledgments ¤

Thanks to Dave Abrahams and Greg Colvin for their comments. Dave and Greg are, with Matt Austern, the authors of the two complete committee proposals for adding these exception safety guarantees into the standard library (see Sidebar 2).¤

## Sidebar 1: Some Standard Helper Functions

The `Stack` and `StackImpl` presented in this article use three helper functions from the standard library: `construct`, `destroy`, and `swap`. In simplified form, here's what these functions look like: ¤

```
// 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);
}
```

This 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 using `delete`. ¤

```
// 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(iter,iter)` is the most interesting. We'll return to it a little later in the main article; it illustrates more than you might think! ¤

To find out more about these standard functions, take a few minutes to examine how they're written in the standard library implementation you're using. It's a very worthwhile and enlightening exercise. ¤

## Sidebar 2: Exception Safety and the Standard Library

Are the standard library containers exception-safe and exception-neutral? The short answer is: Yes. ( 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 be strongly exception-safe.) ¤

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 not to throw — including `swap` (the importance of which was illustrated by my second example), `allocator::deallocate` (the importance of which was illustrated by the discussion of `::operator delete` in the first example) and certain operations of the template parameter types themselves (especially the destructor, the importance of which is illustrated under the subhead "Destructors That Throw and Why They're Evil"). ¤

All standard containers must also implement the strong guarantee for all operations (with two exceptions; see next paragraph): 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: 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 only if `T`'s copy constructor or assignment operator do not throw. ( Unfortunately, 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 afterwards 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 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' 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, and finally use `swap` to switch over to using that new version after you know that the copy-and-change steps have succeeded. ¤