Exception Safety: Concepts and Techniques

Bjarne Stroustrup

AT&T Labs – Research
Florham Park, NJ 07932, USA
`http://www.research.att.com/~bs`

Abstract. This paper presents a set of concepts and design techniques that has proven successful in implementing and using C++ libraries intended for applications that simultaneously require high reliability and high performance. The notion of exception safety is based on the *basic guarantee* that maintains basic invariants and avoids resource leaks and the *strong guarantee* that ensures that a failed operation has no effect.

1 Introduction

This paper, based on *Appendix E: Standard-Library Exception Safety* of *The C++ Programming Language (Special Edition)* [1], presents
(1) a few fundamental concepts useful for discussion of exception safety
(2) effective techniques for crafting exception-safe and efficient containers
(3) some general rules for exception-safe programming.
The discussion of exception safety focuses on the containers provided as part of the ISO C++ standard library [2] [1]. Here, the standard library is used to provide examples of the kinds of concerns that must be addressed in demanding applications. The techniques used to provide exception safety for the standard library can be applied to a wide range of problems.

The discussion assumes an understanding of C++ and a basic understanding of C++'s exception handling mechanisms. These mechanism, the fundamental ways of using them, and the support they receive in the standard library are described in [1]. The reasoning behind the design of C++'s exception handling mechanisms and references to previous work influencing that design can be found in [3].

2 Exception Safety

An operation on an object is said to be *exception safe* if that operation leaves the object in a valid state when the operation is terminated by throwing an exception. This valid state could be an error state requiring cleanup, but it must be well defined so that reasonable error-handling code can be written for the object. For example, an exception handler might destroy the object, repair the object, repeat a variant of the operation, just carry on, etc.

In other words, the object will have an invariant, its constructors establish that invariant, all further operations maintain that invariant, and its destructor does a final cleanup. An operation should take care that the invariant is maintained before throwing an exception, so that the object is in a valid state.

However, it is quite possible for that valid state to be one that doesn't suit the application. For example, a string may have been left as the empty string or a container may have been left unsorted. Thus, ''repair'' means giving an object a value that is more appropriate/desirable for the application than the one it was left with after an operation failed. In the context of the standard library, the most interesting objects are containers.

Here, we consider under which conditions an operation on a standard-library container can be considered exception safe. There can be only two conceptually really simple strategies:

(1) ''*No guarantees*:'' If an exception is thrown, any container being manipulated is possibly corrupted.

(2) ''*Strong guarantee*:'' If an exception is thrown, any container being manipulated remains in the state in which it was before the standard-library operation started.

Unfortunately, both answers are too simple for real use. Alternative (1) is unacceptable because it implies that after an exception is thrown from a container operation, the container cannot be accessed; it can't even be destroyed without fear of run-time errors. Alternative (2) is unacceptable because it imposes the cost of roll-back semantics on every individual standard-library operation.

To resolve this dilemma, the C++ standard library provides a set of exception-safety guarantees that share the burden of producing correct programs between implementers of the standard library and users of the standard library:

(3a) ''*Basic guarantee* for all operations:'' The basic invariants of the standard library are maintained, and no resources, such as memory, are leaked.

(3b) ''*Strong guarantee* for key operations:'' In addition to providing the basic guarantee, either the operation succeeds, or has no effects. This guarantee is provided for key library operations, such as ***push_back***(), single-element ***insert***() on a ***list***, and ***uninitialized_copy***().

(3c) ''*Nothrow guarantee* for some operations:'' In addition to providing the basic guarantee, some operations are guaranteed not to throw an exception This guarantee is provided for a few simple operations, such as ***swap***() and ***pop_back***().

Both the basic guarantee and the strong guarantee are provided on the condition that user-supplied operations (such as assignments and ***swap***() functions) do not leave container elements in invalid states, that user-supplied operations do not leak resources, and that destructors do not throw exceptions.

Violating a standard library requirement, such as having a destructor exit by throwing an exception, is logically equivalent to violating a fundamental language rule, such a dereferencing a null pointer. The practical effects are also equivalent and often disastrous.

In addition to achieving pure exception safety, both the basic guarantee and the strong guarantee ensure the absence of resource leaks. That is, a standard library operation that throws an exception not only leaves its operands in well-defined states but also ensures that every resource that it acquired is (eventually) released. For example, at the point where an exception is thrown, all memory allocated must be either deallocated or owned by some object, which in turn must ensure that the memory is properly deallocated. Remember that memory isn't the only kind of resource that can leak. Files, locks, network connections, and threads are examples of system resources that a function may have to release or hand over to an object before throwing an exception.

Note that the C++ language rules for partial construction and destruction ensure that exceptions thrown while constructing sub-objects and members will be handled correctly without special attention from standard-library code. This rule is an essential underpinning for all techniques dealing with exceptions.

3 Exception-Safe Implementation Techniques

The C++ standard library provides examples of problems that occur in many other contexts and of solutions that apply widely. The basic tools available for writing exception-safe code are

(1) the *try-block*, and

(2) the support for the ''resource acquisition is initialization'' technique.

The key idea behind the ''resource acquisition is initialization'' technique/pattern (sometimes abbreviated to RAII) is that ownership of a resource is given to a scoped object. Typically, that object acquires (opens, allocates, etc.) the resource in its constructor. That way, the objects destructor can release the resource at the end of its life independently of whether that destruction is caused by normal exit from its scope or from an exception. For details, see Sect. 14.4 of [1]. Also, the use of ***vector_base*** from Sect 3.2 of this paper is an example of ''resource acquisition is initialization.''

The general principles to follow are to

(1) don't destroy a piece of information before we can store its replacement

(2) always leave objects in valid states when throwing or re-throwing an exception

(3) avoid resource leaks.

That way, we can always back out of an error situation. The practical difficulty in following these principles is that innocent-looking operations (such as <, =, and ***sort***()) might throw exceptions. Knowing what to look for in an application takes experience.

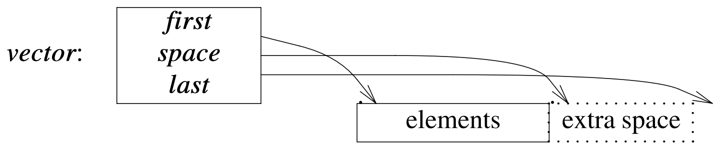
When you are writing a library, the ideal is to aim at the strong exception-safety guarantee and always to provide the basic guarantee. When writing a specific program, there may be less concern for exception safety. For example, if I write a simple data analysis program for my own use, I'm usually quite willing to have the program terminate in the unlikely event of virtual memory exhaustion. However, correctness and exception safety are closely related.

The techniques for providing basic exception safety, such as defining and checking invariants, are similar to the techniques that are useful to get a program small and correct. It follows that the overhead of providing basic exception safety (the basic guarantee) – or even the strong guarantee – can be minimal or even insignificant.

Here, I will consider an implementation of the standard container ***vector*** to see what it takes to achieve that ideal and where we might prefer to settle for more conditional safety.

3.1 A Simple Vector

A typical implementation of *vector* will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (or the equivalent information represented as a pointer plus offsets):



Here is a declaration of *vector* simplified to present only what is needed to discuss exception safety and avoidance of resource leaks:

```
template<class T, class A = allocator<T> > class vector {
public:
        T* v;          // start of allocation
        T* space;      // end of elements, start of space allocated for possible expansion
        T* last;       // end of allocated space
        A alloc;       // allocator

        explicit vector(size_type n, const T& val = T(), const A& = A());
        vector(const vector& a);                // copy constructor
        vector& operator=(const vector& a);     // copy assignment
        ~vector();                              // destructor

        size_type size() const { return space-v; }
        size_type capacity() const { return last-v; }

        void push_back(const T&);

        // ...
};
```

Consider first a naive implementation of a constructor:

```
template<class T, class A>              // warning: naive implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
        :alloc(a)                       // copy the allocator
{
        v = alloc.allocate(n);          // get memory for elements
        space = last = v+n;
        for (T* p=v; p!=last; ++p) a.construct(p,val); // construct copy of val in *p
}
```

There are three sources of exceptions here:

(1) *allocate*() throws an exception indicating that no memory is available;
(2) the allocator's copy constructor throws an exception;
(3) the copy constructor for the element type *T* throws an exception because it can't copy *val*.

In all cases, no object is created. However, unless we are careful, resources can leak.

When *allocate*() fails, the *throw* will exit before any resources are acquired, so all is well. When *T*'s copy constructor fails, we have acquired some memory that must be freed to avoid memory leaks. A more difficult problem is that the copy constructor for *T* might throw an exception after correctly constructing a few elements but before constructing them all. To handle this problem, we could keep track of which elements have been constructed and destroy those (and only those) in case of an error:

```
template<class T, class A>              // elaborate implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
        :alloc(a)                       // copy the allocator
{
        v = alloc.allocate(n);          // get memory for elements

        iterator p;

        try {
                iterator end = v+n;
                for (p=v; p!=end; ++p) alloc.construct(p,val); // construct element
                last = space = p;
        }
        catch (...) {   // destroy constructed elements, free memory, and re-throw:
                for (iterator q = v; q!=p; ++q) alloc.destroy(q);
                alloc.deallocate(v,n);
                throw;
        }
}
```

The overhead here is the overhead of the *try-block*. In a good C++ implementation, this overhead is negligible compared to the cost of allocating memory and initializing elements. For implementations where entering a *try-block* incurs a cost, it may be worthwhile to test *if(n)* before the *try* and handle the empty vector case separately.

The main part of this constructor is an exception-safe implementation of the standard library's *uninitialized_fill()*:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
        For p;
        try {
                for (p=beg; p!=end; ++p)
                        new(static_cast<void*>(&*p)) T(x); // construct copy of x in *p
        }
        catch (...) {   // destroy constructed elements and rethrow:
                for (For q = beg; q!=p; ++q) (&*q)->~T();
                throw;
        }
}
```

The curious construct *&*p* takes care of iterators that are not pointers. In that case, we need to take the address of the element obtained by dereference to get a pointer. The explicit cast to *void** ensures that the standard library placement function is used, and not some user-defined *operator new()* for *T**s. This code is operating at a rather low level where writing truly general code can be difficult.

Fortunately, we don't have to reimplement *uninitialized_fill()*, because the standard library provides the desired strong guarantee for it. It is often essential for initialization to either complete successfully, having initialized every element, or fail leaving no constructed elements behind. Consequently, the standard-library algorithms ***uninitialized_fill***(), ***uninitialized_fill_n***(), and ***uninitialized_copy***() are guaranteed to have this strong exception-safety property.

Note that ***uninitialized_fill***() does not protect against exceptions thrown by element destructors or iterator operations. Doing so would be prohibitively expensive.

The ***uninitialized_fill***() algorithm can be applied to many kinds of sequences. Consequently, it takes a forward iterator and cannot guarantee to destroy elements in the reverse order of their construction.

Using ***uninitialized_fill***(), we can write:

```
template<class T, class A>              // messy implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
        :alloc(a)                               // copy the allocator
{
        v = alloc.allocate(n);                  // get memory for elements
        try {
                uninitialized_fill(v,v+n,val);  // copy elements
                space = last = v+n;
        }
        catch (...) {
                alloc.deallocate(v,n);          // free memory
                throw;                          // re-throw
        }
}
```

However, I wouldn't call that pretty code. The next section will demonstrate how it can be made much simpler.

Note that the constructor re-throws a caught exception. The intent is to make ***vector*** transparent to exceptions so that the user can determine the exact cause of a problem. All standard-library containers have this property. Exception transparency is often the best policy for templates and other ''thin'' layers of software. This is in contrast to major parts of a system (''modules'') that generally need to take responsibility for all exceptions thrown. That is, the implementer of such a module must be able to list every exception that the module can throw. Achieving this may involve grouping exceptions, mapping exceptions from lower-level routines into the module's own exceptions, or exception specification.

3.2 Representing Memory Explicitly

Experience revealed that writing correct exception-safe code using explicit *try-block*s is more difficult than most people expect. In fact, it is unnecessarily difficult because there is an alternative: The ''resource acquisition is initialization'' technique can be used to reduce the amount of code written and to make the code more stylized. In this case, the key resource required by the ***vector*** is memory to hold its elements. By providing an auxiliary class to represent the notion of memory used by a ***vector***, we can simplify the code and decrease the chance of accidentally forgetting to release it:

```
template<class T, class A = allocator<T> >
struct vector_base {
        A alloc;    // allocator
        T* v;       // start of allocation
        T* space;   // end of elements, start of space allocated for possible expansion
        T* last;    // end of allocated space

        vector_base(const A& a, typename A::size_type n)
                : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) { }
        ~vector_base() { alloc.deallocate(v,last-v); }
};
```

As long as *v* and *last* are correct, *vector_base* can be destroyed. Class *vector_base* deals with memory for a type *T*, not objects of type *T*. Consequently, a user of *vector_base* must destroy all constructed objects in a *vector_base* before the *vector_base* itself is destroyed.

Naturally, *vector_base* itself is written so that if an exception is thrown (by the allocator's copy constructor or *allocate*() function) no *vector_base* object is created and no memory is leaked.

We want to be able to *swap*() *vector_base*s. However, the default *swap*() doesn't suit our needs because it copies and destroys a temporary. Because *vector_base* is a special-purpose class that wasn't given fool-proof copy semantics, that destruction would lead to undesirable side effects. Consequently we provide a specialization:

```
template<class T> void swap(vector_base<T>& a, vector_base<T>& b)
{
        swap(a.a,b.a);
        swap(a.v,b.v);
        swap(a.space,b.space);
        swap(a.last,b.last);
}
```

Given *vector_base*, *vector* can be defined like this:

```
template<class T, class A = allocator<T> >
class vector : private vector_base<T,A> {
        void destroy_elements() { for (T* p = v; p!=space; ++p) p->~T(); }
public:
        explicit vector(size_type n, const T& val = T(), const A& = A());
        vector(const vector& a);                 // copy constructor
        vector& operator=(const vector& a);  // copy assignment
        ~vector() { destroy_elements(); }

        size_type size() const { return space-v; }
        size_type capacity() const { return last-v; }

        void push_back(const T&);
        // ...
};
```

The *vector* destructor explicitly invokes the *T* destructor for every element. This implies that if an element destructor throws an exception, the *vector* destruction fails.

This can be a disaster if it happens during stack unwinding caused by an exception and *terminate*() is called. In the case of normal destruction, throwing an exception from a destructor typically leads to resource leaks and unpredictable behavior of code relying on reasonable behavior of objects. There is no really good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws.

Now the constructor can be simply defined:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
        :vector_base<T,A>(a,n)              // allocate space for n elements
{
        uninitialized_fill(v,v+n,val);     // copy elements
}
```

The copy constructor differs by using *uninitialized_copy*() instead of *uninitialized_fill*():

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
        :vector_base<T,A>(a.alloc,a.size())
{
        uninitialized_copy(a.begin(),a.end(),v);
}
```

Note that this style of constructor relies on the fundamental language rule that when an exception is thrown from a constructor, sub-objects (such as bases) that have already been completely constructed will be properly destroyed. The *uninitialized_fill*() algorithm and its cousins provide the equivalent guarantee for partially constructed sequences.

3.3 Assignment

As usual, assignment differs from construction in that an old value must be taken care of. Consider a straightforward implementation:

```
template<class T, class A>              // offers the strong guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
        vector_base<T,A> b(alloc,a.size());             // get memory
        uninitialized_copy(a.begin(),a.end(),b.v);      // copy elements
        destroy_elements();                             // destroy old elements
        alloc.deallocate(v,last-v);                     // free old memory
        vector_base::operator=(b);                      // install new representation
        b.v = 0;                                        // prevent deallocation
        return *this;
}
```

This assignment is nice and exception safe. However, it repeats a lot of code from constructors and destructors. Also, the ‘‘installation’’ of the new *vector_base* is a bit obscure. To avoid this, we can write:

```
template<class T, class A>              // offers the strong guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
        vector temp(a);                                     // copy a
        swap< vector_base<T,A> >(*this,temp);               // swap representations
        return *this;
}
```

The old elements are destroyed by *temp*'s destructor, and the memory used to hold them is deallocated by *temp*'s *vector_base*'s destructor.

The performance of the two versions ought to be equivalent. Essentially, they are just two different ways of specifying the same set of operations. However, the second implementation is shorter and doesn't replicate code from related *vector* functions, so writing the assignment that way ought to be less error prone and lead to simpler maintenance.

Note the absence of the traditional test for self-assignment:

```
        if (this == &a) return *this;
```

These assignment implementations work by first constructing a copy and then swapping representations. This obviously handles self-assignment correctly. I decided that the efficiency gained from the test in the rare case of self-assignment was more than offset by its cost in the common case where a different *vector* is assigned.

In either case, two potentially significant optimizations are missing:

(1) If the capacity of the vector assigned to is large enough to hold the assigned vector, we don't need to allocate new memory.

(2) An element assignment may be more efficient than an element destruction followed by an element construction.

Implementing these optimizations, we get:

```
template<class T, class A>              // optimized, basic guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
        if (capacity() < a.size()) {        // allocate new vector representation:
                vector temp(a);                                     // copy a
                swap< vector_base<T,A> >(*this,temp);               // swap representations
                return *this;
        }

        if (this == &a) return *this;       // protect against self assignment

                                            // assign to old elements:
        size_type sz = size();
        size_type asz = a.size();
        alloc = a.get_allocator();                  // copy the allocator

        if (asz<=sz) {    // copy over old elements and destroy surplus elements:
                copy(a.begin(),a.begin()+asz,v);
                for (T* p = v+asz; p!=space; ++p) p->~T();
        }
        else {        // copy over old elements and construct additional elements:
              copy(a.begin(),a.begin()+sz,v);
              uninitialized_copy(a.begin()+sz,a.end(),space);
        }
        space = v+asz;
        return *this;
}
```

These optimizations are not free. The *copy()* algorithm does *not* offer the strong exception-safety guarantee. It does not guarantee that it will leave its target unchanged if an exception is thrown during copying. Thus, if *T::operator=()* throws an exception during *copy()*, the *vector* being assigned to need not be a copy of the vector being assigned, and it need not be unchanged. For example, the first five elements might be copies of elements of the assigned vector and the rest unchanged. It is also plausible that an element – the element that was being copied when *T::operator=()* threw an exception – ends up with a value that is neither the old value nor a copy of the corresponding element in the vector being assigned. However, if *T::operator=()* leaves its operands in a valid state if it throws an exception, the *vector* is still in a valid state – even if it wasn't the state we would have preferred.

Here, I have copied the allocator using an assignment. It is actually not required that every allocator support assignment.

The standard-library *vector* assignment offers the weaker exception-safety property of this last implementation – and its potential performance advantages. That is, *vector* assignment provides the basic guarantee, so it meets most people's idea of exception safety. However, it does not provide the strong guarantee. If you need an assignment that leaves the *vector* unchanged if an exception is thrown, you must either use a library implementation that provides the strong guarantee or provide your own assignment operation. For example:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // "obvious" a = b
{
        vector<T,A> temp(a.get_allocator());
        temp.reserve(b.size());
        for (typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
              temp.push_back(*p);
        swap(a,temp);
}
```

If there is insufficient memory for *temp* to be created with room for *b.size()* elements, *std::bad_alloc* is thrown before any changes are made to *a*. Similarly, if *push_back()* fails for any reason, *a* will remain untouched because we apply *push_back()* to *temp* rather than to *a*. In that case, any elements of *temp* created by *push_back()* will be destroyed before the exception that caused the failure is re-thrown.

Swap does not copy *vector* elements. It simply swaps the data members of a *vector*; that is, it swaps *vector_base*s (Sect. 3.2). Consequently, it does not throw exceptions even if operations on the elements might. Consequently, *safe_assign()* does not do spurious copies of elements and is reasonably efficient.

As is often the case, there are alternatives to the obvious implementation. We can let the library perform the copy into the temporary for us:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // simple a = b
{
        vector<T,A> temp(b);        // copy the elements of b into a temporary
        swap(a,temp);
}
```

Indeed, we could simply use call-by-value:

```
template<class T, class A>               // simple a = b (note: b is passed by value)
void safe_assign(vector<T,A>& a, vector<T,A> b)
{
        swap(a,b);
}
```

The last two variants of *safe_assign*() don't copy the *vector*'s allocator. This is a permitted optimization.

3.4 *push_back()*

From an exception-safety point of view, *push_back*() is similar to assignment in that we must take care that the *vector* remains unchanged if we fail to add a new element:

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
        if (space == last) {    // no more free space; relocate:
                vector_base b(alloc,size()?2*size():2); // double the allocation
                uninitialized_copy(v,space,b.v);
                new(b.space) T(x);                       // place a copy of x in *b.space
                ++b.space;
                destroy_elements();
                swap<vector_base<T,A> >(b,*this);        // swap representations
                return;
        }
        new(space) T(x);                                 // place a copy of x in *space
        ++space;
}
```

Naturally, the copy constructor initializing **space* might throw an exception. If that happens, the value of the *vector* remains unchanged, with *space* left unincremented. In that case, the *vector* elements are not reallocated so that iterators referring to them are not invalidated. Thus, this implementation implements the strong guarantee that an exception thrown by an allocator or even a user-supplied copy constructor leaves the *vector* unchanged. The standard library offers the strong guarantee for *push_back*().

Note the absence of a *try-block* (except for the one hidden in *uninitialized_copy*()). The update was done by carefully ordering the operations so that if an exception is thrown, the *vector* remains unchanged.

The approach of gaining exception safety through ordering and the ``resource acquisition is initialization'' technique tends to be more elegant and more efficient than explicitly handling errors using *try-block*s. More problems with exception safety arise from a programmer ordering code in unfortunate ways than from lack of specific exception-handling code. The basic rule of ordering is not to destroy information before its replacement has been constructed and can be assigned without the possibility of an exception.

Exceptions introduce possibilities for surprises in the form of unexpected control flows. For a piece of code with a simple local control flow, such as the ***operator***=(), ***safe_assign***(), and ***push_back***() examples, the opportunities for surprises are limited. It is relatively simple to look at such code and ask oneself ''can this line of code throw an exception, and what happens if it does?'' For large functions with complicated control structures, such as complicated conditional statements and nested loops, this can be hard. Adding *try-block*s increases this local control structure complexity and can therefore be a source of confusion and errors. I conjecture that the effectiveness of the ordering approach and the ''resource acquisition is initialization'' approach compared to more extensive use of *try-block*s stems from the simplification of the local control flow. Simple, stylized code is easier to understand and easier to get right.

Note that the ***vector*** implementation is presented as an example of the problems that exceptions can pose and of techniques for addressing those problems. The standard does not require an implementation to be exactly like the one presented here. What the standard does guarantee is described in Sect. E.4 of [1].

3.5 Constructors and Invariants

From the point of view of exception safety, other ***vector*** operations are either equivalent to the ones already examined (because they acquire and release resources in similar ways) or trivial (because they don't perform operations that require cleverness to maintain valid states). However, for most classes, such ''trivial'' functions constitute the majority of code. The difficulty of writing such functions depends critically on the environment that a constructor established for them to operate in. Said differently, the complexity of ''ordinary member functions'' depends critically on choosing a good class invariant. By examining the ''trivial'' ***vector*** functions, it is possible to gain insight into the interesting question of what makes a good invariant for a class and how constructors should be written to establish such invariants.

Operations such as ***vector*** subscripting are easy to write because they can rely on the invariant established by the constructors and maintained by all functions that acquire or release resources. In particular, a subscript operator can rely on *v* referring to an array of elements:

```
template< class T, class A> T& vector<T,A>::operator[](size_type i)
{
      return v[i];
}
```

It is important and fundamental to have constructors acquire resources and establish a simple invariant. To see why, consider an alternative definition of ***vector_base***:

```
template<class T, class A = allocator<T> >        // clumsy use of constructor
class vector_base {
public:
        A alloc;    // allocator
        T* v;       // start of allocation
        T* space;   // end of elements, start of space allocated for possible expansion
        T* last;    // end of allocated space

        vector_base(const A& a, typename A::size_type n)
                : alloc(a), v(0), space(0), last(0)
        {
                v = alloc.allocate(n);
                 space = last = v+n;
        }

        ~vector_base() { if (v) alloc.deallocate(v,last-v); }
};
```

Here, I construct a *vector_base* in two stages: First, I establish a ''safe state'' where *v*, *space*, and *last* are set to *0*. Only after that has been done do I try to allocate memory. This is done out of misplaced fear that if an exception happens during element allocation, a partially constructed object could be left behind. This fear is misplaced because a partially constructed object cannot be ''left behind'' and later accessed. The rules for static objects, automatic objects, member objects, and elements of the standard-library containers prevent that. However, it could/can happen in pre-standard libraries that used/use placement new to construct objects in containers designed without concern for exception safety. Old habits can be hard to break.

Note that this attempt to write safer code complicates the invariant for the class: It is no longer guaranteed that *v* points to allocated memory. Now *v* might be *0*. This has one immediate cost. The standard-library requirements for allocators do not guarantee that we can safely deallocate a pointer with the value *0*. In this, allocators differ from *delete*. Consequently, I had to add a test in the destructor.

This two-stage construct is not an uncommon style. Sometimes, it is even made explicit by having the constructor do only some ''simple and safe'' initialization to put the object into a destructible state. The real construction is left to an *init*() function that the user must explicitly call. For example:

```
template<class T>     // archaic (pre-standard, pre-exception) style
class Vector {
        T* v;       // start of allocation
        T* space;   // end of elements, start of space allocated for possible expansion
        T* last;    // end of allocated space

public:
        Vector() : v(0), space(0), last(0) { }
        ~Vector() { free(v); }

        bool init(size_t n);  // return true if initialization succeeded

        // ... Vector operations ...
};
```

```
template<class T>
bool Vector<T>::init(size_t n) // return true if initialization succeeded
{
        if (v = (T*)malloc(sizeof(T)*n)) {
                uninitialized_fill(v,v+n,T());
                space = last = v+n;
                return true;
        }
        return false;
}
```

The perceived value of this style is

(1) The constructor can't throw an exception, and the success of an initialization using *init*() can be tested by ''usual'' (that is, non-exception) means.
(2) There exists a trivial valid state. In case of a serious problem, an operation can give an object that state.
(3) The acquisition of resources is delayed until a fully initialized object is needed.

However, this two-stage construction technique doesn't deliver its expected benefits and can itself be a source of problems.

The first point (using an *init*() function in preference to a constructor) is bogus. Using constructors and exception handling is a more general and systematic way of dealing with resource acquisition and initialization errors. This style is a relic of pre-exception C++. Having a separate *init*() function is an opportunity to

(1) forget to call *init*(),
(2) call *init*() more than once,
(3) forget to test on the success of *init*(),
(4) forget that *init*() might throw an exception, and
(5) use the object before calling *init*().

Constructors and exceptions were introduced into C++ to prevent such problems [3].

The second point (having an easy-to-construct ''safe'' valid state) is in principle a good one. If we can't put an object into a valid state without fear of throwing an exception before completing that operation, we do indeed have a problem. However, this ''safe state'' should be one that is a natural part of the semantics of the class rather than an implementation artifact that complicates the class invariant.

If the ''safe'' state is not a natural part of the semantics of the class, the invariant is complicated and a burden is imposed on every member function. For example, the simple subscript operation becomes something like:

```
template< class T> T& Vector<T>::operator[](size_type i)
{
        if (v) return v[i];
        // error handling
}
```

If part of the reason for using a two-stage initialization was to avoid exceptions, the error handling part of that *operator*[]() could easily become complicated.

Like the second point, the third (delaying acquisition of a resource until is needed) misapplies a good idea in a way that imposes cost without yielding benefits. In many cases, notably in containers such as *vector*, the best way of delaying resource acquisition is for the programmer to delay the creation of objects until they are needed.

To sum up: the two-phase construction approach leads to more complicated invariants and typically to less elegant, more error-prone, and harder-to-maintain code. Consequently, the language-supported ''constructor approach'' should be preferred to the ''*init*()-function approach'' whenever feasible. That is, resources should be acquired in constructors whenever delayed resource acquisition isn't mandated by the inherent semantics of a class.

The negative effects of two-phase construction become more marked when we consider application classes that acquire significant resources, such as network connections and files. Such classes are rarely part of a framework that guides their use and their implementation in the way the standard-library requirements guide the definition and use of *vector*. The problems also tend to increase as the mapping between the application concepts and the resources required to implement them becomes more complex. Few classes map as directly onto system resources as does *vector*.

4 Implications for Library Users

One way to look at exception safety in the context of the standard library is that we have no problems unless we create them for ourselves: The library will function correctly as long as user-supplied operations meet the standard library's basic requirements. In particular, no exception thrown by a standard container operation will cause memory leaks from containers or leave a container in an invalid state. Thus, the problem for the library user becomes: How can I define my types so that they don't cause undefined behavior or leak resources?

The basic rules are:

(1) When updating an object, don't destroy its old representation before a new representation is completely constructed and can replace the old one without risk of exceptions. For example, see the implementations of *safe_assign*() and *vector*::*push_back*() in Sect. 3.

(1a) If you must override an old representation in the process of creating the new, be sure to leave a valid object behind if an exception is thrown. For example, see the ''optimized'' implementation of *vector*::*operator*=().

(2) Before throwing an exception, release every resource acquired that is not owned by some (other) object.

(2a) The ''resource acquisition is initialization'' technique and the language rule that partially constructed objects are destroyed to the extent that they were constructed can be most helpful here.

(2b) The *uninitialized_copy*() algorithm and its cousins provide automatic release of resources in case of failure to complete construction of a set of objects.

(3) Before throwing an exception, make sure that every operand is in a valid state. That is, leave each object in a state that allows it to be accessed and destroyed without causing undefined behavior or an exception to be thrown from a destructor. For example, see *vector*'s assignment in Sect. 3.2.

(3a) Note that constructors are special in that when an exception is thrown from a constructor, no object is left behind to be destroyed later. This implies that we don't have to establish an invariant and that we must be sure to release all resources acquired during a failed construction before throwing an exception.

(3b) Note that destructors are special in that an exception thrown from a destructor almost certainly leads to violation of invariants and/or calls to ***terminate***().

In practice, it can be surprisingly difficult to follow these rules. The primary reason is that exceptions can be thrown from places where people don't expect them. A good example is ***std***::***bad_alloc***. Every function that directly or indirectly uses ***new*** or an ***allocator*** to acquire memory can throw ***bad_alloc***. In some programs, we can solve this particular problem by not running out of memory. However, for programs that are meant to run for a long time or to accept arbitrary amounts of input, we must expect to handle various failures to acquire resources. Thus, we must assume every function capable of throwing an exception until we have proved otherwise.

One simple way to try to avoid surprises is to use containers of elements that do not throw exceptions (such as containers of pointers and containers of simple concrete types) or linked containers (such as ***list***) that provide the strong guarantee. Another, complementary, approach is to rely primarily on operations, such as ***push_back***(), that offer the strong guarantee that an operation either succeeds or has no effect. However, these approaches are by themselves insufficient to avoid resource leaks and can lead to an ad hoc, overly restrictive, and pessimistic approach to error handling and recovery. For example, a ***vector***<***T****> is trivially exception safe if operations on ***T*** don't throw exceptions. However, unless the objects pointed to are deleted somewhere, an exception from the ***vector*** will lead to a resource leak. Thus, introducing a ***Handle*** class to deal with deallocation and using ***vector***<Handle<T> > rather than the plain ***vector***<***T****> will probably improve the resilience of the code.

When writing new code, it is possible to take a more systematic approach and make sure that every resource is represented by a class with an invariant that provides the basic guarantee. Given that, it becomes feasible to identify the critical objects in an application and provide roll-back semantics (that is, the strong guarantee – possibly under some specific conditions) for operations on such objects.

As mentioned in Sect. 3, the basic techniques for dealing with exceptions, focusing on resources and invariants, also help getting code correct and efficient. In general, keeping code stylish and simple by using classes to represent resources and concepts makes the code easier to understand, easier to maintain, and easier to reason about. Constructors, destructors, and the support for correct partial construction and destruction are the language-level keys to this. ''Resource acquisition is initialization'' is the key programming technique to utilize these language features.

Most applications contain data structures and code that are not written with exception safety in mind. Where necessary, such code can be fitted into an exception-safe framework by either verifying that it doesn't throw exception (as was the case for the C standard library) or through the use of interface classes for which the exception behavior and resource management can be precisely specified.

When designing types intended for use in an exception-safe environment, we must pay special attention to the operations used by the standard library: constructors, destructors, assignments, comparisons, swap functions, functions used as predicates, and operations on iterators. This is best done by defining a class invariant that can be simply established by all constructors. Sometimes, we must design our class invariants so that we can put an object into a state where it can be destroyed even when an operation suffers a failure at an ''inconvenient'' point. Ideally, that state isn't an artifact defined simply to aid exception handling, but a state that follows naturally from the semantics of the type.

When considering exception safety, the emphasis should be on defining valid states for objects (invariants) and on proper release of resources. It is therefore important to represent resources directly as classes. The *vector_base* (Sect. 3.2) is a simple example of this. The constructors for such resource classes acquire lower-level resources (such as the raw memory for *vector_base*) and establish invariants (such as the proper initialization of the pointers of a *vector_base*). The destructors of such classes implicitly free lower-level resources. The rules for partial construction and the ''resource acquisition is initialization'' technique support this way of handling resources.

A well-written constructor establishes the class invariant for an object. That is, the constructor gives the object a value that allows subsequent operations to be written simply and to complete successfully. This implies that a constructor often needs to acquire resources. If that cannot be done, the constructor can throw an exception so that we can deal with that problem before an object is created. This approach is directly supported by the language and the standard library.

The requirement to release resources and to place operands in valid states before throwing an exception means that the burden of exception handling is shared among the function throwing, the functions on the call chain to the handler, and the handler. Throwing an exception does not make handling an error ''somebody else's problem.'' It is the obligation of functions throwing or passing along an exception to release resources that they own and to put operands in consistent states. Unless they do that, an exception handler can do little more than try to terminate gracefully.

5 Acknowledgements

The concepts and techniques described here are the work of many individuals. In particular, Dave Abrahams, Matt Austern, and Greg Colvin made major contributions to the notions of exception safety embodied in the C++ standard library.

6 References

[1] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[2] Andrew Koenig (editor): *Standard – The C++ Language*. ISO/IEC 14882:1998(E). Information Technology Council (NCITS). Washington, DC, USA. http://www.ncits.org/cplusplus.htm.

[3] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. 2000. ISBN 0-201-70073-5.