Appendix E

Standard-Library Exception Safety

Everything will work just as you expect it to,
unless your expectations are incorrect.
– Hyman Rosen

Exception safety — exception-safe implementation techniques — representing resources — assignment — ***push_back***`()` — constructors and invariants — standard container guarantees — insertion and removal of elements — guarantees and tradeoffs — ***swap***`()` — initialization and iterators — references to elements — predicates — ***strings***, streams, algorithms, ***valarray***, and ***complex*** — the C standard library — implications for library users — advice — exercises.

E.1 Introduction

Standard-library functions often invoke operations that a user supplies as function or template arguments. Naturally, some of these user-supplied operations will occasionally throw exceptions. Other functions, such as allocator functions, can also throw exceptions. Consider:

```
void f(vector<X>& v, const X& g)
{
    v[2] = g;                       // X's assignment might throw an exception
    v.push_back(g);                 // vector<X>'s allocator might throw an exception
    sort(v.begin(),v.end());        // X's less-than operation might throw an exception
    vector<X> u = v;                // X's copy constructor might throw an exception
    // ...

    // u destroyed here: we must ensure that X's destructor can work correctly
}
```

What happens if the assignment throws an exception while trying to copy *g*? Will *v* be left with an invalid element? What happens if the constructor that *v.push_back()* uses to copy *g* throws *std::bad_alloc*? Has the number of elements changed? Has an invalid element been added to the container? What happens if *X*'s less-than operator throws an exception during the sort? Have the elements been partially sorted? Could an element have been removed from the container by the sorting algorithm and not put back?

Finding the complete list of possible exceptions in this example is left as an exercise (§E.8[1]). Explaining how this example is well behaved for every well-defined type *X* – even an *X* that throws exceptions – is part of the aim of this appendix. Naturally, a major part of this explanation involves giving meaning and effective terminology to the notions of ''well behaved'' and ''well defined'' in the context of exceptions.

The purpose of this appendix is to

[1] identify how a user can design types that meet the standard library's requirements,

[2] state the guarantees offered by the standard library,

[3] state the standard-library requirements on user-supplied code,

[4] demonstrate effective techniques for crafting exception-safe and efficient containers, and

[5] present a few general rules for exception-safe programming.

The discussion of exception safety necessarily focuses on worst-case behavior. That is, where could an exception cause the most problems? How does the standard library protect itself and its users from potential problems? And, how can users help prevent problems? Please don't let this discussion of exception-handling techniques distract from the central fact that throwing an exception is the best method for reporting an error (§14.1, §14.9). The discussion of concepts, techniques, and standard-library guarantees is organized like this:

§E.2 discusses the notion of exception safety.

§E.3 presents techniques for implementing efficient exception-safe containers and operations.

§E.4 outlines the guarantees offered for standard-library containers and their operations.

§E.5 summarizes exception-safety issues for the non-container parts of the standard library.

§E.6 reviews exception safety from the point of view of a standard-library user.

As ever, the standard library provides 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.

E.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 (§24.3.7.1), its constructors will establish that invariant, all further operations maintain that invariant even if an exception is thrown, and its destructor will do 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 operations on standard-library containers 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***`()` (§E.3.1, §E.4.1).

[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***`()` (§E.4.1).

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. For example, consider these ''handle-like'' (§25.7) classes:

```
template<class T> class Safe {
    T* p;       // p points to a T allocated using new
public:
    Safe() :p(new T) { }
    ~Safe() { delete p; }
    Safe& operator=(const Safe& a) { *p = *a.p; return *this; }
    // ...
};

template<class T> class Unsafe {      // sloppy and dangerous code
    T* p;       // p points to a T
public:
    Unsafe(T* pp) :p(pp) { }
    ~Unsafe() { if (!p->destructible()) throw E(); delete p; }
```

```
        Unsafe& operator=(const Unsafe& a)
        {
                p->~T();                // destroy old value (§10.4.11)
                new(p) T(a.p);          // construct copy of a.p in *p (§10.4.11)
                return *this;
        }

        // ...
};

void f(vector< Safe<Some_type> >&vg, vector< Unsafe<Some_type> >&vb)
{
        vg.at(1) = Safe<Some_type>();
        vb.at(1) = Unsafe<Some_type>(new Some_type);
        // ...
}
```

In this example, construction of a ***Safe*** succeeds only if a ***T*** is successfully constructed. The construction of a ***T*** can fail because allocation might fail (and throw ***std::bad_alloc***) and because ***T***'s constructor might throw an exception. However, in every successfully constructed ***Safe***, ***p*** will point to a successfully constructed ***T***; if a constructor fails, no ***T*** object (or ***Safe*** object) is created. Similarly, ***T***'s assignment operator may throw an exception, causing ***Safe***'s assignment operator to implicitly re-throw that exception. However, that is no problem as long as ***T***'s assignment operator always leaves its operands in a good state. Therefore, ***Safe*** is well behaved, and consequently every standard-library operation on a ***Safe*** will have a reasonable and well-defined result.

On the other hand, ***Unsafe()*** is carelessly written (or rather, it is carefully written to demonstrate undesirable behavior). The construction of an ***Unsafe*** will not fail. Instead, the operations on ***Unsafe***, such as assignment and destruction, are left to deal with a variety of potential problems. The assignment operator may fail by throwing an exception from ***T***'s copy constructor. This would leave a ***T*** in an undefined state because the old value of ****p*** was destroyed and no new value replaced it. In general, the results of that are unpredictable. ***Unsafe***'s destructor contains an ill-conceived attempt to protect against undesirable destruction. However, throwing an exception during exception handling will cause a call of ***terminate()*** (§14.7), and the standard library requires that a destructor return normally after destroying an object. The standard library does not – and cannot – make any guarantees when a user supplies objects this badly behaved.

From the point of view of exception handling, ***Safe*** and ***Unsafe*** differ in that ***Safe*** uses its constructor to establish an invariant (§24.3.7.1) that allows its operations to be implemented simply and safely. If that invariant cannot be established, an exception is thrown before an invalid object is constructed. ***Unsafe***, on the other hand, muddles along without a meaningful invariant, and the individual operations throw exceptions without an overall error-handling strategy. Naturally, this results in violations of the standard library's (reasonable) assumptions about the behavior of types. For example, ***Unsafe*** can leave invalid elements in a container after throwing an exception from ***T::operator=()*** and may throw an exception from its destructor.

Note that the standard-library guarantees relative to ill-behaved user-supplied operations are analogous to the language guarantees relative to violations of the basic type system. If a basic operation is not used according to its specification, the resulting behavior is undefined. For

example, if you throw an exception from a destructor for a ***vector*** element, you have no more reason to hope for a reasonable result than if you dereference a pointer initialized to a random number:

```
class Bomb {
public:
    // ...
    ~Bomb() { throw Trouble(); };
};

vector<Bomb> b(10);                          // leads to undefined behavior

void f()
{
    int* p = reinterpret_cast<int*>(rand());  // leads to undefined behavior
    *p = 7;
}
```

Stated positively: If you obey the basic rules of the language and the standard library, the library will behave well even when you throw exceptions.

In addition to achieving pure exception safety, we usually prefer to avoid resource leaks. That is, an operation that throws an exception should not only leave its operands in well-defined states but also ensure 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.

The standard-library guarantees the absence of resource leaks provided that user-supplied operations called by the library also avoid resource leaks. Consider:

```
void leak(bool abort)
{
    vector<int> v(10);                              // no leak
    vector<int>* p = new vector<int>(10);           // potential memory leak
    auto_ptr< vector<int> > q(new vector<int>(10)); // no leak (§14.4.2)

    if (abort) throw Up();
    // ...
    delete p;
}
```

Upon throwing the exception, the ***vector*** called ***v*** and the ***vector*** held by ***q*** will be correctly destroyed so that their resources are released. The ***vector*** pointed to by ***p*** is not guarded against exceptions and will not be destroyed. To make this piece of code safe, we must either explicitly delete ***p*** before throwing the exception or make sure it is owned by an object – such as an ***auto_ptr*** (§14.4.2) – that will properly destroy it if an exception is thrown.

Note that the 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 (§14.4.1). This rule is an essential underpinning for all techniques dealing with exceptions.

Also, remember that memory isn't the only kind of resource that can leak. Opened 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.

E.3 Exception-Safe Implementation Techniques

As usual, the 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* (§8.3.1), and

[2] the support for the ''resource acquisition is initialization'' technique (§14.4).

The general principles to follow are to

[3] never let go of a piece of information before we can store its replacement, and

[4] always leave objects in valid states when throwing or re-throwing an exception.

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.

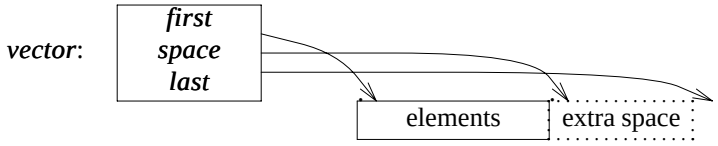
When you are writing a library, the ideal is to aim at the strong exception-safety guarantee (§E.2) and always to provide the basic guarantee. When writing a specific program, there may be less concern for exception safety. For example, if I write a simple data analysis program for my own use, I'm usually quite willing to have the program terminate in the unlikely event of virtual memory exhaustion. However, correctness and basic exception safety are closely related.

The techniques for providing basic exception safety, such as defining and checking invariants (§24.3.7.1), 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; §E.2) – or even the strong guarantee – can be minimal or even insignificant; see §E.8[17].

Here, I will consider an implementation of the standard container *vector* (§16.3) to see what it takes to achieve that ideal and where we might prefer to settle for more conditional safety.

E.3.1 A Simple Vector

A typical implementation of *vector* (§16.3) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§17.1.3) (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 {
private:
      T* v;       // start of allocation
      T* space;   // end of element sequence, start of space allocated for possible expansion
      T* last;    // end of allocated space
      A alloc;    // allocator
```

```
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();

    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>
vector<T,A>::vector(size_type n, const T& val, const A& a)  // warning: naive implementation
    :alloc(a)                                               // copy the allocator
{
    v = alloc.allocate(n);                                  // get memory for elements (§19.4.1)
    space = last = v+n;
    for (T* p = v; p!=last; ++p) a.construct(p,val);        // construct copy of val in *p (§19.4.1)
}
```

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, so ***vector***'s destructor is not called (§14.4.1).

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>
vector<T,A>::vector(size_type n, const T& val, const A& a)    // elaborate implementation
    :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 (§19.4.1)
        last = space = p;
    }
```

```
        catch (...) {
                for (iterator q = v; q!=p; ++q) alloc.destroy(q);     // destroy constructed elements
                alloc.deallocate(v,n);                                // free memory
                throw;                                                // re-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 add a 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 **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 (§10.4.11)
        }
        catch (...) {  // destroy constructed elements and rethrow:
                for (For q = beg; q!=p; ++q) (&*q)->~T();  // (§10.4.11)
                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 (§19.4.5), 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 (§E.2). It is often essential to have initialization operations that 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**() (§19.4.4) are guaranteed to have this strong exception-safety property (§E.4.4).

Note that the **uninitialized_fill**() algorithm does not protect against exceptions thrown by element destructors or iterator operations (§E.4.4). Doing so would be prohibitively expensive (see §E.8[16-17]).

The **uninitialized_fill**() algorithm can be applied to many kinds of sequences. Consequently, it takes a forward iterator (§19.2.1) and cannot guarantee to destroy elements in the reverse order of their construction.

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

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)    // messy implementation
    :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 (§14.2), mapping exceptions from lower-level routines into the module's own exceptions (§14.6.3), or exception specification (§14.6).

E.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 (§14.4) can be used to reduce the amount of code needing to be 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 element sequence, 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 destructions would lead to undesirable sideeffects. 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(); } // §10.4.11
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 (§14.7). 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 (§E.4).

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 (§14.4.1). The ***uninitialized_fill()*** algorithm and its cousins (§E.4.4) provide the equivalent guarantee for partially constructed sequences.

E.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>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // offers the strong guarantee (§E.2)
{
    vector_base<T,A> b(alloc,a.size());          // get memory
    uninitialized_copy(a.begin(),a.end(),b.v);   // copy elements
    destroy_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 safe, but it repeats a lot of code from constructors and destructors. To avoid this, we could write:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // offers the strong guarantee (§E.2)
{
    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 (§10.4.4). 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>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // optimized, basic guarantee (§E.2)
{
    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 (§10.4.4)

                                      // assign to old elements:
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();                         // copy the allocator
    if (asz<=sz) {
        copy(a.begin(),a.begin()+asz,v);
        for (T* p = v+asz; p!=space; ++p) p->~T();     // destroy surplus elements (§10.4.11)
    }
    else {
        copy(a.begin(),a.begin()+sz,v);
        uninitialized_copy(a.begin()+sz,a.end(),space);    // construct extra elements
    }
    space = v+asz;
    return *this;
}
```

These optimizations are not free. The *copy()* algorithm (§18.6.1) 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 (§19.4.3); see also §E.8[9].

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 (§E.2). 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 (§E.3.2). Consequently, it does not throw exceptions even if operations on the elements might (§E.4.3). 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 (§7.2):

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

E.3.4 *push_back()*

From an exception-safety point of view, ***push_back()*** is similar to the 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 (§10.4.11)
        ++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 (§10.4.11)
    ++space;
}
```

Naturally, the copy constructor used to initialize ***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 that guarantee for ***push_back()*** (§E.4.1).

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 (§14.4) 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 (§14.4). 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 the subject of §E.4.

E.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 (§24.3.7.1). 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 (§16.3.3) 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 element sequence, 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 (§10.4.11) 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* (§19.4.1). In this, allocators differ from ***delete*** (§6.2.6). Consequently, I had to add a test in the destructor. Also, each element is first initialized and then assigned. The cost of doing that extra work can be significant for element types for which assignment is nontrivial, such as ***string*** and ***list***.

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_base {
public:
      T* v;       // start of allocation
      T* space;   // end of element sequence, start of space allocated for possible expansion
      T* last;    // end of allocated space

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

      bool 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 actually needed.

The following subsections examine these points and shows why this two-stage construction technique doesn't deliver its expected benefits. It can also be a source of problems.

E.3.5.1 Using *init()* Functions

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 (§14.1, §14.4). This style is a relic of pre-exception C++.

Carefully written code using the two styles are roughly equivalent. Consider:

```
int f1(int n)
{
    vector<X> v;
    // ...
    if (v.init(n)) {
        // use v as vector of n elements
    }
    else {
        // handle_problem
    }
}
```

and

```
int f2(int n)
try {
    vector v<X> v(n);
    // ...
    // use v as vector of n elements
}
catch (...) {
    // handle problem
}
```

However, having a separate ***init()*** function is an opportunity to
[1] forget to call ***init()*** (§10.2.3),
[2] forget to test on the success of ***init()***,
[3] call ***init()*** more than once,
[4] forget that ***init()*** might throw an exception, and
[5] use the object before calling ***init()***.
The definition of ***vector<T>::init()*** illustrates [4].

In a good C++ implementation, ***f2()*** will be marginally faster than ***f1()*** because it avoids the test in the common case.

E.3.5.2 Relying on a Default Valid State

The second point (having an easy-to-construct ''default'' valid state) is correct in general, but in the case of ***vector***, it is achieved at an unnecessary cost. It is now possible to have a ***vector_base*** with ***v==0***, so the *vector* implementation must protect against that possibility throughout. For example:

```
template< class T>
T& vector<T>::operator[](size_t i)
{
    if (v) return v[i];
    // handle error
}
```

Leaving the possibility of ***v==0*** open makes the cost of non-range-checked subscripting equivalent to range-checked access:

```
template< class T>
T& vector<T>::at(size_t i)
{
    if (i<v.size()) return v[i];
    throw out_of_range("vector index");
}
```

What fundamentally happened here was that I complicated the basic invariant for ***vector_base*** by introducing the possibility of ***v==0***. In consequence, the basic invariant for ***vector*** was similarly complicated. The end result of this is that all code in ***vector*** and ***vector_base*** must be more complicated to cope. This is a source of potential errors, maintenance problems, and run-time overhead. Note that conditional statements can be surprisingly costly on modern machine architectures. Where efficiency matters, it can be crucial to implement a key operation, such as vector subscripting, without conditional statements.

Interestingly, the original definition of ***vector_base*** already did have an easy-to-construct valid state. No ***vector_base*** object could exist unless the initial allocation succeeded. Consequently, the implementer of ***vector*** could write an ''emergency exit'' function like this:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
    space = v;                  // set the size of *this to 0
    throw Total_failure();
}
```

This is a bit drastic because it fails to call element destructors and to deallocate the space for elements held by the ***vector_base***. That is, it fails to provide the basic guarantee (§E.2). If we are willing to trust the values of ***v*** and ***space*** and the element destructors, we can avoid potential resource leaks:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
    destroy_elements();         // clean up
    throw Total_failure();
}
```

Please note that the standard ***vector*** is such a clean design that it minimizes the problems caused by two-phase construction. The ***init()*** function is roughly equivalent to ***resize()***, and in most places the possibility of ***v==0*** is already covered by ***size()==0*** tests. The negative effects described for 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***.

The idea of having a ''safe 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.

E.3.5.3 Delaying resource acquisition

Like the second point (§E.3.5.2), the third (to delay acquisition until a resource 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. Consider a naive use of ***vector***:

```
void f(int n)
{
    vector<X> v(n);      // make n default objects of type X
    // ...

    v[3] = X(99);        // real ''initialization'' of v[3]
    // ...
}
```

Constructing an ***X*** only to assign a new value to it later is wasteful – especially if an ***X*** assignment is expensive. Therefore, two-phase construction of ***X*** can seem attractive. For example, the type ***X*** may itself be a ***vector***, so we might consider two-phase construction of ***vector*** to optimize creation of empty ***vector***s. However, creating default (empty) vectors is already efficient, so complicating the implementation with a special case for the empty vector seems futile. More generally, the best solution to spurious initialization is rarely to remove complicated initialization from the element constructors. Instead, a user can create elements only when needed. For example:

```
void f2(int n)
{
    vector<X> v;              // make empty vector
    // ...

    v.push_back(X(99));       // construct element when 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.

E.4 Standard Container Guarantees

If a library operation itself throws an exception, it can – and does – make sure that the objects on which it operates are left in a well-defined state. For example, ***at***() throwing ***out_of_range*** for a ***vector*** (§16.3.3) is not a problem with exception safety for the ***vector***. The writer of ***at***() has no problem making sure that a ***vector*** is in a well-defined state before throwing. The problems – for

library implementers, for library users, and for people trying to understand code – come when a user-supplied function throws an exception.

The standard-library containers offer the basic guarantee (§E.2): The basic invariants of the library are maintained, and no resources are leaked as long as user code behaves as required. That is, user-supplied operations should not leave container elements in invalid states or throw exceptions from destructors. By ''operations,'' I mean operations used by the standard-library implementation, such as constructors, assignments, destructors, and operations on iterators (§E.4.4).

It is relatively easy for the programmer to ensure that such operations meet the library's expectations. In fact, much naively written code conforms to the library's requirements. The following types clearly meet the standard library's requirements for container element types:

[1] built-in types – including pointers,
[2] types without user-defined operations,
[3] classes with operations that neither throw exceptions nor leave operands in invalid states,
[4] classes with destructors that don't throw exceptions and for which it is simple to verify that operations used by the standard library (such as constructors, assignments, `<`, `==`, and *swap*()) don't leave operands in invalid states.

In each case, we must also make sure that no resource is leaked. For example:

```
void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
    vector<Shape*> v(10);          // either create vector or throw bad_alloc
    v[3] = pc;                     // no exception thrown
    v.insert(v.begin()+4,pt);      // either insert pt or no effect on v
    v2.erase(v2.begin()+3);        // either erase v2[3] or no effect on v2
    v2 = v;                        // copy v or no effect on v2
    // ...
}
```

When *f*() exits, *v* will be properly destroyed, and *v2* will be in a valid state. This fragment does not indicate who is responsible for deleting *pc* and *pt*. If *f*() is responsible, it can either catch exceptions and do the required deletion, or assign the pointers to local *auto_ptr*s.

The more interesting question is: When do the library operations offer the strong guarantee that an operation either succeeds or has no effect on its operands? For example:

```
void f(vector<X>& vx)
{
    vx.insert(vx.begin()+4,X(7));      // add element
}
```

In general, *X*'s operations and *vector<X>*'s allocator can throw an exception. What can we say about the elements of *vx* when *f*() exits because of an exception? The basic guarantee ensures that no resources have been leaked and that *vx* has a set of valid elements. However, exactly what elements? Is *vx* unchanged? Could a default *X* have been added? Could an element have been removed because that was the only way for *insert*() to recover while maintaining the basic guarantee? Sometimes, it is not enough to know that a container is in a good state; we also want to know exactly what state that is. After catching an exception, we typically want to know that the elements are exactly those we intended, or we will have to start error recovery.

E.4.1 Insertion and Removal of Elements

Inserting an element into a container and removing one are obvious examples of operations that might leave a container in an unpredictable state if an exception is thrown. The reason is that insertions and deletions invoke many operations that may throw exceptions:

[1] A new value is copied into a container.
[2] An element deleted from (erased from) a container must be destroyed.
[3] Sometimes, memory must be allocated to hold a new element.
[4] Sometimes, ***vector*** and ***deque*** elements must be copied to new locations.
[5] Associative containers call comparison functions for elements.
[6] Many insertions and deletions involve iterator operations.

Each of these cases can cause an exception to be thrown.

If a destructor throws an exception, no guarantees are made (§E.2). Making guarantees in this case would be prohibitively expensive. However, the library can and does protect itself – and its users – from exceptions thrown by other user-supplied operations.

When manipulating a linked data structure, such as a ***list*** or a ***map***, elements can be added and removed without affecting other elements in the container. This is not the case for a container implemented using contiguous allocation of elements, such as a ***vector*** or a ***deque***. There, elements sometimes need to be moved to new locations.

In addition to the basic guarantee, the standard library offers the strong guarantee for a few operations that insert or remove elements. Because containers implemented as linked data structures behave differently from containers with contiguous allocation of elements, the standard provides slightly different guarantees for different kinds of containers:

[1] Guarantees for ***vector*** (§16.3) and ***deque*** (§17.2.3):
- If an exception is thrown by a ***push_back***( ) or a ***push_front***( ), that function has no effect.
- Unless thrown by the copy constructor or the assignment operator of the element type, if an exception is thrown by an ***insert***( ), that function has no effect.
- Unless thrown by the copy constructor or the assignment operator of the element type, no ***erase***( ) throws an exception.
- No ***pop_back***( ) or ***pop_front***( ) throws an exception.

[2] Guarantees for ***list*** (§17.2.2):
- If an exception is thrown by a ***push_back***( ) or a ***push_front***( ), that function has no effect.
- If an exception is thrown by an ***insert***( ), that function has no effect.
- No ***erase***( ), ***pop_back***( ), ***pop_front***( ), ***splice***( ), or ***reverse***( ) throws an exception.
- Unless thrown by a predicate or a comparison function, the ***list*** member functions ***remove***( ), ***remove_if***( ), ***unique***( ), ***sort***( ), and ***merge***( ) do not throw exceptions.

[3] Guarantees for associative containers (§17.4):
- If an exception is thrown by an ***insert***( ) while inserting a single element, that function has no effect.
- No ***erase***( ) throws an exception.

Note that where the strong guarantee is provided for an operation on a container, all iterators, pointers to elements, and references to elements remain valid if an exception is thrown.

These rules can be summarized in a table:

Container-Operation Guarantees				
	vector	**deque**	**list**	**map**
clear()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
erase()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
1-element insert()	strong (*copy*)	strong (*copy*)	strong	strong
N-element insert()	strong (*copy*)	strong (*copy*)	strong	basic
merge()	—	—	nothrow (*comparison*)	—
push_back()	strong	strong	strong	—
push_front()	—	strong	strong	—
pop_back()	nothrow	nothrow	nothrow	—
pop_front()	—	nothrow	nothrow	—
remove()	—	—	nothrow (*comparison*)	—
remove_if()	—	—	nothrow (*predicate*)	—
reverse()	—	—	nothrow	—
splice()	—	—	nothrow	—
swap()	nothrow	nothrow	nothrow	nothrow (*copy-of-comparison*)
unique()	—	—	nothrow (*comparison*)	—

In this table:

basic means that the operation provides only the basic guarantee (§E.2)
strong means that the operation provides the strong guarantee (§E.2)
nothrow means that the operation does not throw an exception (§E.2)
— means that the operation is not provided as a member of this container

Where a guarantee requires that some user-supplied operations not throw exceptions, those operations are indicated in parentheses under the guarantee. These requirements are precisely stated in the text preceding the table.

The *swap*() functions differ from the other functions mentioned by not being members. The guarantee for *clear*() is derived from that offered by *erase*() (§16.3.6). This table lists guarantees offered in addition to the basic guarantee. Consequently this table does not list operations, such as *reverse*() and *unique*() for *vector*, that are provided only as algorithms for all sequences without additional guarantees.

The ''almost container'' *basic_string* (§17.5, §20.3) offers the basic guarantee for all operations (§E.5.1). The standard also guarantees that *basic_string*'s *erase*() and *swap*() don't throw, and offers the strong guarantee for *basic_string*'s *insert*() and *push_back*().

In addition to ensuring that a container is unchanged, an operation providing the strong guarantee also leaves all iterators, pointers, and references valid. For example:

```
void update(map<string,X>& m, map<string,X>::iterator current)
{
    X x;
    string s;
    while (cin>>s>>x)
    try {
        current = m.insert(current,make_pair(s,x));
    }
    catch(...) {
        // here current still denotes the current element
    }
}
```

E.4.2 Guarantees and Tradeoffs

The patchwork of additional guarantees reflects implementation realities. Programmers prefer the strong guarantee with as few conditions as possible, but they also tend to insist that each individual standard-library operation be optimally efficient. Both concerns are reasonable, but for many operations, it is not possible to satisfy both simultaneously. To give a better idea of the tradeoffs involved, I'll examine ways of adding of single and multiple elements to *list*s, *vector*s, and *map*s.

Consider adding a single element to a *list* or a *vector*. As ever, *push_back*() provides the simplest way of doing that:

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
    try {
        lst.push_back(x);          // add to list
    }
    catch (...) {
        // lst is unchanged
        return;
    }
```

```
    try {
        vec.push_back(x);          // add to vector
    }
    catch (...) {
        // vec is unchanged
        return;
    }
    // lst and vec each have a new element with the value x
}
```

Providing the strong guarantee in these cases is simple and cheap. It is also very useful because it provides a completely exception-safe way of adding elements. However, *push_back()* isn't defined for associative containers – a *map* has no *back()*. After all, the last element of an associative container is defined by the order relation rather than by position.

The guarantees for *insert()* are a bit more complicated. The reason is that sometimes *insert()* has to place an element in ''the middle'' of a container. This is no problem for a linked data structure, such as *list* or *map*. However, if there is free reserved space in a *vector*, the obvious implementation of *vector<X>::insert()* copies the elements after the insertion point to make room. This is optimally efficient, but there is no simple way of restoring a *vector* if *X*'s copy assignment or copy constructor throws an exception (see §E.8[10-11]). Consequently, *vector* provides a guarantee that is conditional upon element copy operations not throwing exceptions. However, *list* and *map* don't need such a condition; they can simply link in new elements after doing any necessary copying.

As an example, assume that *X*'s copy assignment and copy constructor throw *X::cannot_copy* if they cannot successfully create a copy:

```
void f(list<X>& lst, vector<X>& vec, map<string,X>& m, const X& x, const string& s)
{
    try {
        lst.insert(lst.begin(),x);          // add to list
    }
    catch (...) {
        // lst is unchanged
        return;
    }

    try {
        vec.insert(vec.begin(),x);          // add to vector
    }
    catch (X::cannot_copy) {
        // oops: vec may or may not have a new element
        return;
    }
    catch (...) {
        // vec is unchanged
        return;
    }
```

```
        try {
                m.insert(make_pair(s,x));       // add to map
        }
        catch (...) {
                // m is unchanged
                return;
        }
        // lst and vec each have a new element with the value x
        // m has an element with the value (s,x)
}
```

If ***X*::*cannot_copy*** is caught, a new element may or may not have been inserted into ***vec***. If a new element was inserted, it will be an object in a valid state, but it is unspecified exactly what the value is. It is possible that after ***X*::*cannot_copy***, some element will have been ''mysteriously'' duplicated (see §E.8[11]). Alternatively, ***insert*()** may be implemented so that it deletes some ''trailing'' elements to be certain that no invalid elements are left in a container.

Unfortunately, providing the strong guarantee for ***vector***'s ***insert*()** without the caveat about exceptions thrown by copy operations is not feasible. The cost of completely protecting against an exception while moving elements in a ***vector*** would be significant compared to simply providing the basic guarantee in that case.

Element types with copy operations that can throw exceptions are not uncommon. Examples from the standard library itself are ***vector<string>***, ***vector< vector<double> >***, and ***map<string,int>***.

The ***list*** and ***vector*** containers provide the same guarantees for ***insert*()** of single and multiple elements. The reason is simply that for ***vector*** and ***list***, the same implementation strategies apply to both single-element and multiple-element ***insert*()**. However, ***map*** provides the strong guarantee for single-element ***insert*()**, but only the basic guarantee for multiple-element ***insert*()**. A single-element ***insert*()** for ***map*** that provides the strong guarantee is easily implemented. However, the obvious strategy for implementing multiple-element ***insert*()** for a ***map*** is to insert the new elements one after another, and it is not easy to provide the strong guarantee for that. The problem with this is that there is no simple way of backing out of previous successful insertions if the insertion of an element fails.

If we want an insertion function that provides the strong guarantee that either every element was successfully added or the operation had no effect, we can build it by constructing a new container and then ***swap*()**:

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
        C tmp(c.begin(),i);                             // copy leading elements to temporary
        copy(begin,end,inserter(tmp,tmp.end()));        // copy new elements
        copy(i,c.end(),inserter(tmp,tmp.end()));        // copy trailing elements
        swap(c,tmp);
}
```

As ever, this code may misbehave if the element destructor throws an exception. However, if an element copy operation throws an exception, the argument container is unchanged.

E.4.3 Swap

Like copy constructors and assignments, *swap()* operations are essential to many standard algorithms and are often supplied by users. For example, *sort()* and *stable_sort()* typically reorder elements, using *swap()*. Thus, if a *swap()* function throws an exception while exchanging values from a container, the container could be left with unchanged elements or a duplicate element rather than a pair of swapped elements.

Consider the obvious definition of the standard-library *swap()* function (§18.6.8):

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Clearly, *swap()* doesn't throw an exception unless the element type's copy constructor or copy assignment does.

With one minor exception for associative containers, standard container *swap()* functions are guaranteed not to throw exceptions. Basically, containers are swapped by exchanging the data structures that act as handles for the elements (§13.5, §17.1.3). Since the elements themselves are not moved, element constructors and assignments are not invoked, so they don't get an opportunity to throw an exception. In addition, the standard guarantees that no standard-library *swap()* function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. This leaves only one potential source of exceptions: The comparison object in an associative container is copied as part of the handle. The only possible exception from a *swap()* of standard containers is the copy constructor and assignment of the container's comparison object (§17.1.4.1). Fortunately, comparison objects usually have trivial copy operations that do not have opportunities to throw exceptions.

A user-supplied *swap()* should be written to provide the same guarantees. This is relatively simple to do as long as one remembers to swap types represented as handles by swapping their handles, rather than slowly and elaborately copying the information referred to by the handles (§13.5, §16.3.9, §17.1.3).

E.4.4 Initialization and Iterators

Allocation of memory for elements and the initialization of such memory are fundamental parts of every container implementation (§E.3). Consequently, the standard algorithms for constructing objects in uninitialized memory – *uninitialized_fill()*, *uninitialized_fill_n()*, and *uninitialized_copy()* (§19.4.4) – are guaranteed to leave no constructed objects behind if they throw an exception. They provide the strong guarantee (§E.2). This sometimes involves destroying elements, so the requirement that destructors not throw exceptions is essential to these algorithms; see §E.8[14]. In addition, the iterators supplied as arguments to these algorithms are required to be well behaved. That is, they must be valid iterators, refer to valid sequences, and iterator operations (such as ++ and != and *) on a valid iterator are not allowed to throw exceptions.

Iterators are examples of objects that are copied freely by standard algorithms and operations on standard containers. Thus, copy constructors and copy assignments of iterators should not throw exceptions. In particular, the standard guarantees that no copy constructor or assignment operator of an iterator returned from a standard container throws an exception. For example, an iterator returned by *vector<T>::begin()* can be copied without fear of exceptions.

Note that ++ and -- on an iterator can throw exceptions. For example, an *istreambuf_iterator* (§19.2.6) could reasonably throw an exception to indicate an input error, and a range-checked iterator could throw an exception to indicate an attempt to move outside its valid range (§19.3). However, they cannot throw exceptions when moving an iterator from one element of a sequence to another, without violating the definition of ++ and -- on an iterator. Thus, *uninitialized_fill()*, *uninitialized_fill_n()*, and *uninitialized_copy()* assume that ++ and -- on their iterator arguments will not throw; if they do throw, either those ''iterators'' weren't iterators according to the standard, or the ''sequence'' specified by them wasn't a sequence. Again, the standard containers do not protect the user from the user's own undefined behavior (§E.2).

E.4.5 References to Elements

When a reference, a pointer, or an iterator to an element of a container is handed to some code, that code can corrupt the container by corrupting the element. For example:

```
void f(const X& x)
{
    list<X> lst;
    lst.push_back(x);
    list<X>::iterator i = lst.begin();
    *i = x;      // copy x into list
    // ...
}
```

If *x* is corrupted, *list*'s destructor may not be able to properly destroy *lst*. For example:

```
struct X {
    int* p;

    X() { p = new int; }
    ~X() { delete p; }
    // ...
};

void malicious()
{
    X x;
    x.p = reinterpret_cast<int*>(7);      // corrupt x
    f(x);                                 // time bomb
}
```

When the execution reaches the end on *f()*, the *list<X>* destructor is called, and that will in turn invoke *X*'s destructor for the corrupted value. The effect of executing *delete p* when *p* isn't *0* and doesn't point to an *X* is undefined and could be an immediate crash. Alternatively, it

might leave the free store corrupted in a way that causes difficult-to-track problems much later on in an apparently unrelated part of a program.

This possibility of corruption should not stop people from manipulating container elements through references and iterators; it is often the simplest and most efficient way of doing things. However, it is wise to take extra care with such references into containers. When the integrity of a container is crucial, it might be worthwhile to offer safer alternatives to less experienced users. For example, we might provide an operation that checks the validity of a new element before copying it into an important container. Naturally, such checking can only be done with knowledge of the application types.

In general, if an element of a container is corrupted, subsequent operations on the container can fail in nasty ways. This is not particular to containers. Any object left in a bad state can cause subsequent failure.

E.4.6 Predicates

Many standard algorithms and many operations on standard containers rely on predicates that can be supplied by users. In particular, all associative containers depend on predicates for both lookup and insertion.

A predicate used by a standard container operation may throw an exception. In that case, every standard-library operation provides the basic guarantee, and some operations, such as *insert*() of a single element, provide the strong guarantee (§E.4.1). If a predicate throws an exception from an operation on a container, the resulting set of elements in the container may not be exactly what the user wanted, but it will be a set of valid elements. For example, if == throws an exception when invoked from *list*::*unique*() (§17.2.2.3), the user cannot assume that no duplicates are in the list. All the user can safely assume is that every element on the list is valid (see §E.5.3).

Fortunately, predicates rarely do anything that might throw an exception. However, user-defined <, ==, and != predicates must be taken into account when considering exception safety.

The comparison object of an associative container is copied as part of a *swap*() (§E.4.3). Consequently, it is a good idea to ensure that the copy operations of predicates that might be used as comparison objects do not throw exceptions.

E.5 The Rest of the Standard Library

The crucial issue in exception safety is to maintain the consistency of objects; that is, we must maintain the basic invariants for individual objects and the consistency of collections of objects. In the context of the standard library, the objects for which it is the most difficult to provide exception safety are the containers. From the point of view of exception safety, the rest of the standard library is less interesting. However, note that from the perspective of exception safety, a built-in array is a container that might be corrupted by an unsafe operation.

In general, standard-library functions throw only the exceptions that they are specified to throw, plus any thrown by user-supplied operations that they may call. In addition, any function that (directly or indirectly) allocates memory can throw an exception to indicate memory exhaustion (typically, *std*::*bad_alloc*).

E.5.1 Strings

The operations on ***string***s can throw a variety of exceptions. However, ***basic_string*** manipulates its characters through the functions provided by ***char_traits*** (§20.2), and these functions are not allowed to throw exceptions. That is, the ***char_traits*** supplied by the standard library do not throw exceptions, and no guarantees are made if an operation of a user-defined ***char_traits*** throws an exception. In particular, note that a type used as the element (character) type for a ***basic_string*** is not allowed to have a user-defined copy constructor or a user-defined copy assignment. This removes a significant potential source of exception throws.

A ***basic_string*** is very much like a standard container (§17.5, §20.3). In fact, its elements constitute a sequence that can be accessed using ***basic_string<Ch,Tr,A>::iterator***s and ***basic_string<Ch,Tr,A>::const_iterator***s. Consequently, a string implementation offers the basic guarantee (§E.2), and the guarantees for ***erase***(), ***insert***(), ***push_back***() and ***swap***() (§E.4.1) apply to ***basic_string***s. For example, ***basic_string<Ch,Tr,A>::push_back***() offers the strong guarantee.

E.5.2 Streams

If required to do so, iostream functions throw exceptions to signal state changes (§21.3.6). The semantics of this are well defined and pose no exception-safety problems. If a user-defined ***operator<<***() or ***operator>>***() throws an exception, it may appear to the user as if the iostream library threw an exception. However, such an exception will not affect the stream state (§21.3.3). Further operations on the stream may not find the expected data – because the previous operation threw an exception instead of completing normally – but the stream itself is uncorrupted. As ever after an I/O problem, a ***clear***() may be needed before doing further reads/writes (§21.3.3, §21.3.5).

Like ***basic_string***, the *iostream*s rely on ***char_traits*** to manipulate characters (§20.2.1, §E.5.1). Thus, an implementation can assume that operations on characters do not throw exceptions, and no guarantees are made if the user violates that assumption.

To allow for crucial optimizations, ***locale***s (§D.2) and ***facet***s (§D.3) are assumed not to throw exceptions. If they do, a stream using them could be corrupted. However, the most likely exception, a ***std::bad_cast*** from a ***use_facet*** (§D.3.1), can occur only in user-supplied code outside the standard stream implementation. At worst, this will produce incomplete output or cause a read to fail rather than corrupt the ***ostream*** (or ***istream***) itself.

E.5.3 Algorithms

Aside from ***uninitialized_copy***(), ***uninitialized_fill***(), and ***uninitialized_fill_n***() (§E.4.4), the standard offers only the basic guarantee (§E.2) for algorithms. That is, provided that user-supplied objects are well behaved, the algorithms will maintain all standard-library invariants and leak no resources. To avoid undefined behavior, user-supplied operations should always leave their operands in valid states, and destructors should not throw exceptions.

The algorithms themselves do not throw exceptions. Instead, they report errors and failures through their return values. For example, search algorithms generally return the end of a sequence to indicate ‘‘not found’’ (§18.2). Thus, exceptions thrown from a standard algorithm

must originate from a user-supplied operation. That is, the exception must come from an operation on an element – such as a predicate (§18.4), an assignment, or a ***swap***( ) – or from an allocator (§19.4).

If such an operation throws an exception, the algorithm terminates immediately, and it is up to the functions that invoked the algorithm to handle the exception. For some algorithms, it is possible for an exception to occur at a point where the container is not in a state that the user would consider good. For example, some sorting algorithms temporarily copy elements into a buffer and later put them back into the container. Such a ***sort***( ) might copy elements out of the container (planning to write them back in proper order later), overwrite them, and then throw an exception. From a user's point of view, the container was corrupted. However, all elements are in a valid state, so recovery should be reasonably straightforward.

Note that the standard algorithms access sequences through iterators. That is, the standard algorithms never operate on containers directly, only on elements in a container. The fact that a standard algorithm never directly adds or removes elements from a container simplifies the analysis of the impact of exceptions. Similarly, if a data structure is accessed only through iterators, pointers, and references to ***const*** (for example, through a ***const Rec****), it is usually trivial to verify that an exception has no undesired effects.

E.5.4 Valarray and Complex

The numeric functions do not explicitly throw exceptions (Chapter 22). However, ***valarray*** needs to allocate memory and thus might throw ***std*::*bad_alloc***. Furthermore, ***valarray*** or ***complex*** may be given an element type (scalar type) that throws exceptions. As ever, the standard library provides the basic guarantee (§E.2), but no specific guarantees are made about the effects of a computation terminated by an exception.

Like ***basic_string*** (§E.5.1), ***valarray*** and ***complex*** are allowed to assume that their template argument type does not have user-defined copy operations so that they can be bitwise copied. Typically, these standard-library numeric types are optimized for speed, assuming that their element type (scalar type) does not throw exceptions.

E.5.5 The C Standard Library

A standard-library operation without an exception specification may throw exceptions in an implementation-defined manner. However, functions from the standard C library do not throw exceptions unless they take a function argument that does. After all, these functions are shared with C, and C doesn't have exceptions. An implementation may declare a standard C function with an empty *exception-specification*, ***throw***( ), to help the compiler generate better code.

Functions such as ***qsort***( ) and ***bsearch***( ) (§18.11) take a pointer to function as argument. They can therefore throw an exception if their arguments can. The basic guarantee (§E.2) covers these functions.

E.6 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 (§E.2). 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 ***vector*::*operator*=()**, ***safe_assign*()**, and ***vector*::*push_back*()** in §E.3.

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

[2a] The ''resource acquisition is initialization'' technique (§14.4) and the language rule that partially constructed objects are destroyed to the extent that they were constructed (§14.4.1) can be most helpful here. For example, see ***leak*()** in §E.2.

[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 (§E.4.4).

[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 §E.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 (§E.4). 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 (§E.2). 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 (§25.7) 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 (§E.2). 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.

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 exceptions (as was the case for the C standard library; §E.5.5) 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 (§E.3.5).

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*** (§E.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 (§14.4.1) and the ''resource acquisition is initialization'' technique (§14.4) support this way of handling resources.

A well-written constructor establishes the class invariant for an object (§24.3.7.1). 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 (§E.3.5).

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.

E.7 Advice

[1] Be clear about what degree of exception safety you want; §E.2.
[2] Exception safety should be part of an overall strategy for fault tolerance; §E.2.
[3] Provide the basic guarantee for all classes. That is, maintain an invariant, and don't leak resources; §E.2, §E.3.2, §E.4.
[4] Where possible and affordable, provide the strong guarantee that an operation either succeeds or leaves all operands unchanged; §E.2, §E.3.
[5] Don't throw an exception from a destructor; §E.2, §E.3.2, §E.4.
[6] Don't throw an exception from an iterator navigating a valid sequence; §E.4.1, §E.4.4.
[7] Exception safety involves careful examination of individual operations; §E.3.
[8] Design templates to be transparent to exceptions; §E.3.1.
[9] Prefer the constructor approach to resource requisition to using ***init***`()` functions; §E.3.5.
[10] Define an invariant for a class to make it clear what is a valid state; §E.2, §E.6.
[11] Make sure that an object can always be put into a valid state without fear of an exception being thrown; §E.3.2, §E.6.
[12] Keep invariants simple; §E.3.5.
[13] Leave all operands in valid states before throwing an exception; §E.2, §E.6.
[14] Avoid resource leaks; §E.2, §E.3.1, §E.6.
[15] Represent resources directly; §E.3.2, §E.6.
[16] Remember that ***swap***`()` can sometimes be an alternative to copying elements; §E.3.3.
[17] Where possible, rely on ordering of operations rather than on explicit use of *try-block*s; §E.3.4.
[18] Don't destroy ''old'' information until its replacement has been safely produced; §E.3.3, §E.6.
[19] Rely on the ''resource acquisition is initialization'' technique; §E.3, §E.3.2, §E.6.
[20] Make sure that comparison operations for associative containers can be copied; §E.3.3.
[21] Identify critical data structures and provide them with operations that provide the strong guarantee; §E.6

E.8 Exercises

1. (∗1) List all exceptions that could possibly be thrown from ***f***`()` in §E.1.
2. (∗1) Answer the questions after the example in §E.1.
3. (∗1) Define a class ***Tester*** that occasionally throws exceptions from basic operations, such as copy constructors. Use ***Tester*** to test your standard-library containers.
4. (∗1) Find the error in the ''messy'' version of ***vector***'s constructor (§E.3.1), and write a program to get it to crash. Hint: First implement ***vector***'s destructor.
5. (∗2) Implement a simple list providing the basic guarantee. Be very specific about what the list requires of its users to provide the guarantee.
6. (∗3) Implement a simple list providing the strong guarantee. Carefully test this list. Give an argument why people should believe it to be safe.
7. (∗2.5) Reimplement ***String*** from §11.12 to be as safe as a standard container.
8. (∗2) Compare the run time of the various versions of ***vector***'s assignment and

safe_assign`()` (§E.3.3).

9. (∗1.5) Copy an allocator without using an assignment operator (as needed to improve ***operator=***`()` in §E.3.3).
10. (∗2) Add single-element and multiple-element ***erase***`()` and ***insert***`()` that provide the basic guarantee to ***vector*** (§E.3.2).
11. (∗2) Add single-element and multiple-element ***erase***`()` and ***insert***`()` that provide the strong guarantee to ***vector*** (§E.3.2). Compare the cost and complexity of these solutions to the solutions to exercise 10.
12. (∗2) Write a ***safe_insert***`()` (§E.4.2) that inserts elements into the existing ***vector*** (rather than copying to a temporary). What constraints do you have to impose on operations?
13. (∗2) Write a ***safe_insert***`()` (§E.4.2) that inserts elements into the existing ***map*** (rather than copying to a temporary). What constraints do you have to impose on operations?
14. (∗2.5) Compare the size, complexity, and performance of the ***safe_insert***`()` functions from exercises 12 and 13 to the ***safe_insert***`()` from §E.4.2.
15. (∗2.5) Write a better (simpler and faster) ***safe_insert***`()` for associative containers only. Use traits to write a ***safe_insert***`()` that automatically selects the optimal ***safe_insert***`()` for a container. Hint: §19.2.3.
16. (∗2.5) Try to rewrite ***uninitialized_fill***`()` (§19.4.4, §E.3.1) to handle destructors that throw exceptions. Is that possible? If so, at what cost? If not, why not?
17. (∗2.5) Try to rewrite ***uninitialized_fill***`()` (§19.4.4, §E.3.1) to handle iterators that throw exceptions for `++` and `--`. Is that possible? If so, at what cost? If not, why not?
18. (∗3) Take a container from a library different from the standard library. Examine its documentation to see what exception-safety guarantees it provides. Do some tests to see how resilient it is against exceptions thrown by memory allocation and user-supplied code. Compare it to a corresponding standard-library container.
19. (∗3) Try to optimize the ***vector*** from §E.3 by disregarding the possibility of exceptions. For example, remove all *try-block*s. Compare the performance against the version from §E.3 and against a standard-library ***vector*** implementation. Also, compare the size and the complexity of the source code of these different ***vector***s.
20. (∗1) Define invariants for ***vector*** (§E.3) with and without the possibility of ***v==0*** (§E.3.5) .
21. (∗2.5) Read the source of an implementation of ***vector***. What guarantees are implemented for assignment, multi-element ***insert***`()`, and ***resize***`()`?
22. (∗3) Write a version of ***hash_map*** (§17.6) that is as safe as a standard container.