Standard-Library Exception safety

Bjarne Stroustrup

AT&T Labs – Research

http://www.research.att.com/~bs

bs@research.att.com

Abstract

Designing containers and algorithms that are simultaneously efficient and exception safe is hard. The C++ standard library was designed to meet this challenge. This talk presents the guarantees offered by the standard, the requirements on user code that these requirements depend on, and explains the reasons behind the tradeoffs. Examples from a standard-library vector implementation are used to illustrate general techniques. As ever, the standard library provides examples of widely applicable techniques and principles.

Standard-Library Exception safety

- The problem
- The general approach in the standard library
- Examples: aspects of a vector implementation
- A few general rules/suggestions

Further reading:

- Appendix E of "The C++ Programming Language"
  - 3rd edition or "special edition"
  - on my home pages (http://www.research.att.com/~bs)
- Sutter: "Exceptional C++"
  - C++ In-depth series, Addison Wesley

Exception safety

- A library implementation must call user-supplied functions that might throw exceptions

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

- Will it be safe to use **v** after this call?
  - Can **v** be destroyed after this call?
  - (what is the complete set of possible exceptions that might be thrown?)

Exception safety

- Not every piece of code require the same degree of fault tolerance
  - Here, I focus on programs with stringent reliability requirements
- The standard library must be usable in essentially every program
  - Thus, it provides good examples
- Concepts, design approaches and techniques are more important than details
  - But you can’t understand principles without examples

Exception safety guarantees

- Simple/fundamental approaches:
  - Full rollback semantics (operations succeed or have no effects)
    - too expensive for most uses of containers and often not needed
  - No exception guarantees
    - precludes cleanup after an exception has been thrown
- The standard provides a (reasonable) set of guarantees
  - Requirements share responsibility between library implementer and library user
  - Complexity is a result of addressing practical needs
    - Predictability and efficiency needs

Exception guarantees

- Basic guarantee (for all operations)
  - The basic library invariants are maintained
  - No resources (such as memory) are leaked
- Strong guarantee (for some key operations)
  - Either the operation succeeds or it has no effects
- No throw (for some key operations)
  - The operation does not throw an exception

Provided that destructors do not throw exceptions

- Further requirements for individual operations

Exception guarantees

- For a user-supplied operation to violate a standard-library requirement is like a user-supplied operation performing an undefined operation:
  - All bets are off
  - Just don’t do it (easy to say, often also easy to do)
- For example:
  - A destructor for an element throws an exception
  - A copy operation leaves a container element in an invalid state
  - A function dereferences a random number

Exception guarantees
Example: list<>::push_back()

```
void f(list<X>& lst, X g)
{
    try {
        lst.push_back(g);
    }
    catch (…) {
        // exception thrown (directly or indirectly) by push_back()
        // the strong guarantee: lst is unchanged
        return;
    }
    // lst has a new element with the same value as g
}
```

Exception guarantees
Example: vector<>::push_back()

```
void f(vector<X>& v, X g)
{
    try {
        v.push_back(g);
    }
    catch (X::cannot_copy) {
        // exception thrown by X's copy constructor
        //  The basic guarantee : v is not corrupted,
        //                        but we don't know exactly what elements v has
    }
    catch (…) {
        // exception thrown (directly or indirectly) by push_back()
        // the strong guarantee: v is unchanged
        return;
    }
    // v has a new element with the same value as g
}
```

Exception guarantees

Note:

- Even the basic guarantee ensures “no resource leaks”
  - Resource management is a key concern for error handling
- All guarantees are conditional on “reasonable behavior”
  - Invariants (valid states) is a key concern for error handling

Exception guarantees

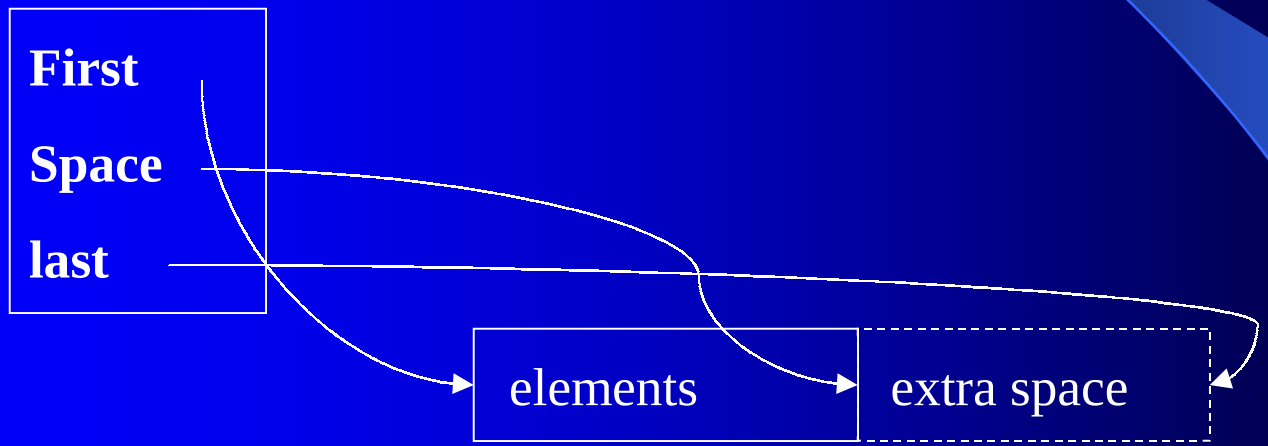
- Fault tolerance isn’t just catching all exceptions
  - The program must be in a valid state after an exception is caught
  - Resource leaks must be avoided
- Good error handling requires cooperation
  - The thrower
  - Functions on the rewind path
  - The handler

Exception guarantees

- Keys to practical exception safety
  - Partial construction handled correctly by the language
  - “Resource acquisition is initialization” technique
  - Define and maintain invariants for important types

Exception safety: vector

vector:



Exception safety: vector

```
template<class T, class A = allocator<T> > class vector {
    T* v;               // start of allocation
    T* space;           // end of element sequence, start of free space
    T* last;            // end of allocation
    A alloc;            // allocator
public:
    // …
    vector(const vector&);                  // copy constructor
    vector& operator=(const vector&);       // copy assignment
    void push_back(const T&);               // add element at end
    size_type size() const { return space-v; } // calculated, not stored
};
```

Unsafe constructor (1)

- Leaks memory
  - but does **not** create bad vectors

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

Unititialized_fill()

- offers the strong guarantee:

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& val)
{
    For p;
    try {
        for (p=beg; p!=end; ++p) new(&*p) T(val);        // construct
    }
    catch (…) {                                          // undo construction
        for (For q = beg; q!=p; ++q) q->~T();            // destroy
        throw;                                           // rethrow
    }
}
```

Unsafe constructor (2)

- Better, but it still leaks memory

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                                   // copy allocator
{
    v = a.allocate(n);                          // get memory for elements
    space = last = unintialized_fill(v,v+n,val);  // copy val into elements
}
```

Represent memory explicitly

```
template<class T, class A> class vector_base {    // manage space
public:
    A& alloc;       // allocator
    T* v;           // start of allocated space
    T* space;       // end of element sequence, start of free space
    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); }
};
```

A vector provides access to memory

```
template<class T, class A = allocator<T> >
class vector : private vector_base {
    void destroy_elements() { for(T* p = v; p!=space; ++p) p->~T(); }
public:
    // …
    explicit vector(size_type n, const T& v = T(), const A& a = A());
    vector(const vector&);                  // copy constructor
    vector& operator=(const vector&);       // copy assignment
    ~vector() { destroy_elements(); }
    void push_back(const T&);               // add element at end
    size_type size() const { return space-v; } // calculated, not stored
    // …
};
```

Exception safety: vector

- Given **vector_base** we can write simple **vector** constructors that don't leak

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

Exception safety: vector

- Given **vector_base** we can write simple **vector** constructors that don't leak

```
template<class T, class A >
vector<T,A>::vector(const vector& a)
    : vector_base(a.get_allocator(),a.size()) // allocate space for a.size() elements
{
    uninitialized_copy(a.begin(),a.end(),v);          // initialize
}
```

Exception safety: vector

- Native assignment (unsafe)

```
template<class T, class A = allocator<A> >
Vector<T,A>& Vector<T,A>::operator=(const vector& a)
{
    destroy_elements();                          // destroy old elements
    alloc.deallocate(v);                         // free old allocation
    alloc = a.get_allocator();                   // copy allocator
    v = alloc.allocate(a.size());                // allocate
    for (int i = 0; i<a.size(); i++) v[i] = a.v[i];   // copy elements
    space = last = v+a.size();
    return *this;
}
```

Naïve Assignment w. strong guarantee

- Construct new value, **then** destroy the old one

```
template<class T, class A = allocator<A> >
Vector<T,A>& Vector<T,A>::operator=(const vector& a)
{
    vector_base<T,A> b(alloc,a.size());          // get space for new vector
    uninitialized_copy(a.begin(),a.end(),b.v);
    destroy_elements();                          // destroy old elements
    allcoc.deallocate(v,last-v);                 // free old memory
    vector_base::operator=(b);                   // install new representation
    b.v = 0;
    return *this;
}
```

Assignment w. strong guarantee

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

- **Note:**
  - **The algorithm is not optimal**
    - **What if the new value fits in the old allocation?**
  - **The implementation is optimal**
  - **No check for self assignment (not needed)**
  - **The "naïve" assignment simply duplicated code from other parts of the vector implementation**

Optimized assignment (1)

```
template<class T, class A = allocator<A> >
Vector<T,A>& Vector<T,A>::operator=(const vector& a)
{
    if (capacity() < a.size()) {          // allocate new vector representation
        vector temp(a);
        swap< vector_base<T,A> >(*this,temp);
        return *this;
    }
    if (this == &a) return *this;         // self assignment
    //  copy into existing space
    return *this;
}
```

Optimized assignment (2)

```
template<class T, class A = allocator<A> >
Vector<T,A>& Vector<T,A>::operator=(const vector& a)
{
    // …
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();
    if (asz<=sz) {
        copy(a.begin(),a.begin()+asz,v);
        for (T* p =v+asz; p!=space; ++p) p->~T();       // destroy surplus elements
    }
    else {
        copy(a.begin(),a.begin()+sz,v);
        uninitialized_copy(a.begin()+sz,a.end(),space); // construct extra elements
    }
    space = v+asz;
    return *this;
}
```

Optimized assignment (3)

- The optimized assignment
  - 19 lines of code
    - 3 lines for the unoptimized version
  - offers the basic guarantee
    - not the strong guarantee
  - can be an order of magnitude faster than the unoptimized version
    - depends on usage and on free store manager
  - is what the standard library offers
    - I.e. only the basic guarantee is offered
    - But your implementation may differ and provide a stronger guarantee

Safe Vector Assignment

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b)
{
    vector<T,A> temp(b);
    swap(a,temp);
}
```

- Vector **swap()** is optimized so that it doesn't copy elements
  - And it doesn't throw exceptions

Safe Container Assignment

```
template<class C> void safe_assign(C& a, const C& b)
{
    C  temp(b);
    swap(a,temp);
}

// Or even:

template<class C> void safe_assign(C& a, const C b)      // call by value
{
    swap(a,b);
}
```

Invariants, Constructors, and Exceptions

- What makes a good class invariant?
  - Simple
  - Can be established by constructor
  - Makes member functions simple
  - Can always be re-established before throwing an exception

Invariants, Constructors, and Exceptions

- A good invariant makes member functions simple:

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

- We don’t need to check for **v!=0**
  - if the constructor could not allocate and initialize the vector, no vector is constructed

Invariants, Constructors, and Exceptions

- Consider an alternative:

```
template<class T> class vector {          // archaic, pre-exception style
        T *v, *space, *last;
        vector() { v = space = last = 0; }    // safe default state
        ~vector() { delete v;  }
        void init(size_t n) { v = new T[n]; space = last = v+n; }
        bool valid() { return v!=0; }         // test that init() succeeded
        // …
};
```

- Perceived value:
  - The constructor can't throw an exception
  - We can test that init() succeeded by traditional (i.e. non-exception) means
  - There is a trivial "safe" state
  - Resource acquisition is delayed until a fully initialized object is needed

Invariants, Constructors, and Exceptions

- Having a separate **init()** function is an opportunity to
  - Forget to call **init()**
  - Forget to test that **init()** succeeded
  - Forget that **init()** might throw an exception
  - Use an object before calling **init()**
- Constructors and exceptions were invented to avoid such problems

Invariants, Constructors, and Exceptions

- **init()** functions complicate invariants
  - and that complicates functions

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

  - In this case, the complexity of unchecked access became equivalent to the complexity of checked access

Invariants, Constructors, and Exceptions

- Delayed acquisition
  - Don’t define an object before you need it
  - Provide suitable semantics (e.g. **vector::resize()**)

Invariants, Constructors, and Exceptions

- A “simple safe state” can usually be provided without complicating the invariant

```
template<class T, class A>
void vector<T,A>::emergency_exit()
{
    destroy_elements();      // or “space=v” if you’re paranoid
    throw Total_failure();
}
```

Container Guarantees
(Slightly Abbreviated)

	erase	1-insert	N-insert	merge	push_back	push_front	remove	swap
vector	nothrow (copy)	strong (copy)	strong (copy)	--	strong	--	--	nothrow
deque	nothrow (copy)	strong (copy)	strong (copy)	--	strong	strong	--	nothrow
list	nothrow	strong	strong	nothrow (comparison)	strong	strong	nothrow	nothrow
map	nothrow	strong	basic	--	--	--	-- (copy-of-comparison)	nothrow

Exception safety

- Rules of thumb:
  - Decide which level of fault tolerance you need
    - Not all code needs to be exception safe
  - Aim at providing the strong guarantee and (always) provide the basic guarantee if you can't afford the strong guarantee
  - Define "good state" (invariant) carefully
    - Establish the invariant in constructors (not in "**init()** functions")
  - Always keep a good state (usually the old state) until you have constructed a new state; then update "atomically"
  - Represent resources directly
    - Prefer "resource acquisition is initialization" over code where possible
  - Minimize explicit try blocks
  - Keep code highly structured ("stylized")
    - "random code" easily hides exception problems