Standard-Library Exception Safety

Bjarne Stroustrup

AT&T Labs – Research

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

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.

Also: Introduction to the C++ exception handling mechanisms and “resource acquisition is initialization” for people with little experience with exceptions

90 mins. + Q&A

Standard-library exception safety

- The problem
- C++ exception-handling mechanisms and concepts
- 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 Handling

- The problem:

  provide a systematic way of handling run-time errors

  - C and C++ programmers use many traditional techniques
    - Error return values, error functions, error state, …
    - Chaos in programs composed out of separately-developed parts
  - Traditional techniques do not integrate well with C++
    - Errors in constructors
    - Errors in composite objects

Exception Handling

- General idea for dealing with non-local errors:
  - Caller knows (in principle) how to handle an error
    - But cannot detect it (or else if would be a local error)
  - Callee can detect an error
    - But does not know how to handle it
  - Let a caller express interest in a type of error

```
try {
        // do work
} catch (Error) {
        // handle error
}
```

  - Let a callee exit with an indication of a kind of error
    - **throw Error();**

Exception handling

```
class Size_error { };
class Range_error { };

class Vector {
    // …
    Vector(int s) { if (s<0 || max_size<=s) throw Size_error(); /* … */ }
    int& operator[](int s) { if (s<0 || size()<=s) throw Range_error(); /* … */ }
};

void f(int x)
{
    Vector v(300000);
    v[x] = 7;
}
catch(Size_error) { cerr << “Oops, size error in f()”; }
catch(Range_error) { cerr << “Oops, range error in f()”; }
```

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
    FILE* f = fopen(p,”r”);      // acquire
    // use f
    fclose(f);                   // release
 }
```

Managing Resources

```
//      naïve fix:

void f(const char* p)
{
      FILE* f = 0;
      try {
        f = fopen(p,”r”);
        // use f
      }
      catch (…) {            // handle exception
        // …
      }
      if (f) fclose(f);
  }
```

Managing Resources

```
// use an object to represent a resource (“resource acquisition in initialization”)

class File_handle {      // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r) { p = fopen(pp,r); }
        File_handle(const string& s, const char* r) { p = fopen(s.c_str(),r); }
        ~File_handle() { if (p) fclose(p); }  // destructor
        // copy operations
        // access functions
};

void f(string s)
{
        File_handle f(s,”r”);
        // use f
}
```

Invariants

- To recover from an error we must leave our program in a “good state”
- Each class has a notion of what is its “good state”
  - Called its invariant
- An invariant is established by a constructor

```
class Vector {
    int sz;
    int* elem;  // elem points to an array of sz ints
public:
    vector(int s) :sz(s), elem(new int(s)) { }  // I’ll discuss error handling elsewhere
    // …
};
```

Exception safety

- Not every individual piece of code require the same degree of fault tolerance
  - Here, I focus on programs with stringent reliability requirements
  - Remember: good error handling is multi-level
- 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 guarantee (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

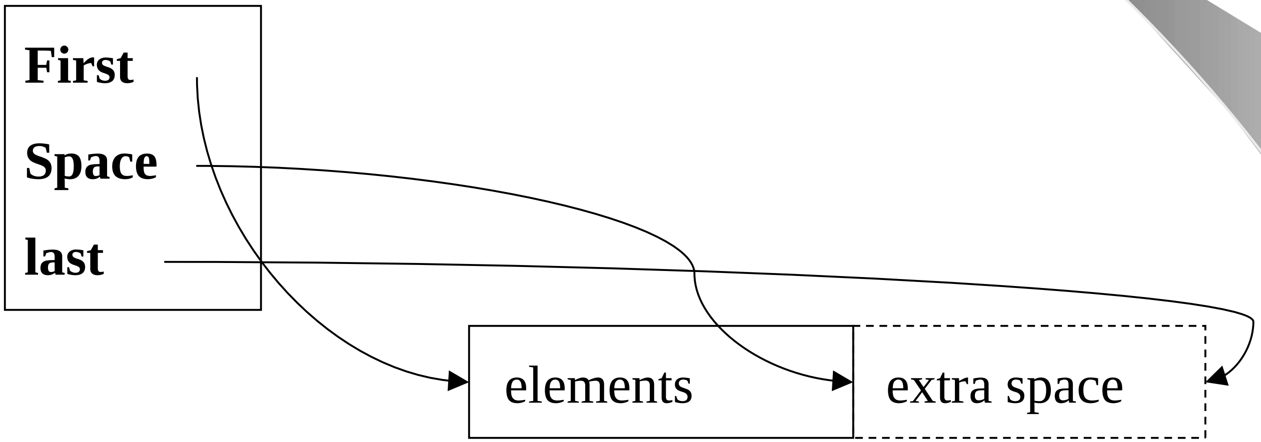
- Fault tolerance isn’t just catching all exceptions
  - The program must be in a valid state after an exception is caught
  - Resource leaks must be avoided
- Good error handling requires cooperation
  - The thrower
  - Functions on the rewind path
  - The handler

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 and other resources
  - 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 = uninitialized_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 is something that 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

- Naïve assignment (unsafe)

```
template<class T, class 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 with strong guarantee

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

```
template<class T, class 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
    swap< vector_base<T,A> >swap(*this,b);       // install new representation
    return *this;
}
```

Assignment with strong guarantee

```
template<class T, class 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>
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 >
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 (“nothrow guarantee”)

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?

    (from an exception-handling point of view)

    - 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()**
  - Call **init()** twice
  - 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 (how?)
}
```

  - 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 (copy)	--	--	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 every individual piece of 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
    - Keep a good state (usually the old state) until you have constructed a new state; then update “atomically”
  - Define “good state” (invariant) carefully
    - Establish the invariant in constructors (not in “**init()** functions”)
  - Minimize explicit try blocks
  - Represent resources directly
    - Prefer “resource acquisition is initialization” over code where possible
    - Avoid “free standing” **new**s and **delete**s
  - Keep code highly structured (“stylized”)
    - “random code” easily hides exception problems