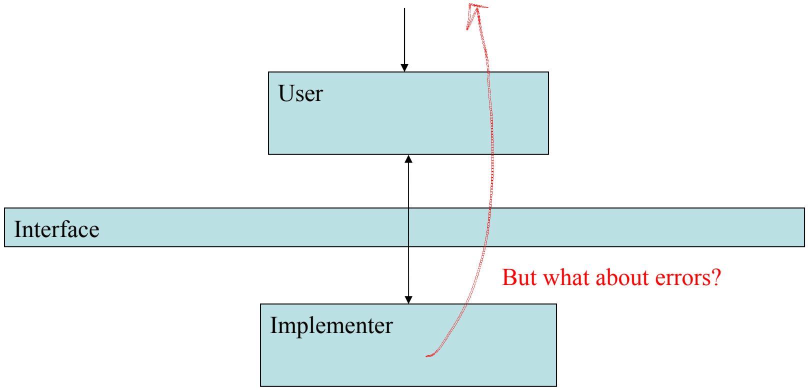
# Handling Errors using Exceptions

Bjarne Stroustrup
bs@cs.tamu.edu
http://www.research.att.com/~bs

---

# Interfaces

- *interface* is the central concept in programming


User
Interface
Implementer
But what about errors?

---

# How do we report run-time errors?

- Error state

```
int r = f(x,y);       // may set errno (yuck!)
if (errno) { /* do something */ }
```

- Error return codes

```
int r = area(lgt,w);         // can return any positive int
if (r<=0) { /* do something */ }
int r = f(x,y);              // f() can return any int (bummer!)
```

- (error-code,value) pairs

```
pair<Error_no,int> rr = are
if (rr.first) { /* do something */ }
pnt r = rr.second;           // good value
```

- Exceptions

```
try { int a = area(lgt,w); /* … */ }
catch(Bad_area){ /* do something */ }
```

---

# 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
  - Code using exceptions can be really elegant
    - And efficient

# 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();**

---

# 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); if (p==0) throw Bad_file(); }
        File_handle(const string& s, const char* r)
                { p = fopen(s.c_str(),r); if (p==0) throw Bad_file(); }
        ~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
    // …
};
```

# Invariants

- An invariant is established by a constructor
  - It may acquire resources
- A good invariant makes it easier to write member functions
- All resources owned must be returned by the destructor

```
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
    ~vector() { delete[] elem; }
    // …
};
```

# RAII: Resource Acquisition Is Initialization

- In the C++ standard library
  - All containers: **vector, list, map, unordered_map, set**, …
  - **string**
  - Regular expressions; **regex, submatch**
  - Iostreams: **fstream, stringstream**
    - manage their buffers and device connections
  - **thread**
  - **lock_guard, unique_lock**
  - "Smart pointers": **unique_ptr, shared_ptr**

# 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
  - Like a database transaction
- 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

- Keys to practical exception safety
  - Partial construction handled correctly by the language

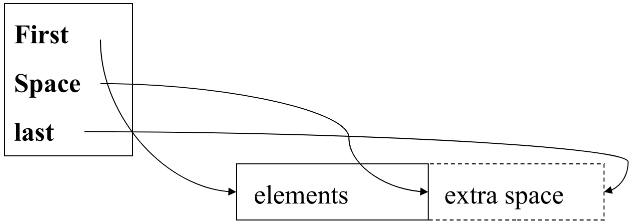
```
class X { X(int); /* … */ };
class Y { Y(int); /* … */ };
class Z { Z(int); /* … */ };
class D : X, Y { Y m1; Z m2; D(int); /*
  … */ };
```

  - “Resource acquisition is initialization” technique
  - Define and maintain invariants for important types

---

# Exception safety: vector

**vector**:



Best vector<T>() representation seems to be (0,0,0)

---

# 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(size_type n, const T& val, const A& a); // constructor
    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
    size_type capacity() const { return last-space; }
};
```

---

# 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); }
};

// works best if a.allocate(0)==0
// we have assumed a stored allocator for convenience
```

---

# 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
}
```

---

# But how do you handle errors?

- Where do you catch?
  - Multi-level?
- Did you remember to catch?
  - Static vs. dynamic vs. no checking

---

# Reserve() is key

- That's where most of the tricky memory management reside

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;          // never decrease allocation
    vector_base<T,A> b(alloc,newalloc);          // allocate new space
    for (int i=0; i<sz; ++i) alloc.construct(&b.elem[i],elem[i]);     // copy
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i],space);              //
    destroy old
    swap< vector_base<T,A> >(*this,b);          // swap representations
}
```

# Push_back() is (now) easy

```
template<class T, class A>
void vector<T,A>::push_back(const T& val)
{
    if (sz==capacity()) reserve(sz?2*space:4);        // get more space
    alloc.construct(&elem[sz],d);                      // add d at end
    ++sz;                                              // increase the size
}
```

# Resize()

- Similarly, **vector<T,A>::resize()** is not too difficult:

```
template<class T, class A>
void vector<T,A>::resize(int newsize, T val = T())
{
   reserve(newsize);
   for (int i=sz; i<newsize; ++i) alloc.construct(&elem[i],val); // construct
   for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]); // destroy
   sz = newsize;
}
```

# 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;
}
```

# 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

# 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
  - 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

# Exceptions and threads

- You can transfer an exception from one thread to another, e.g.:

```
promise<X> px;
try {
        X res;
        // …
        px.set_value(res);
}
catch(…) {
        px.set_exception(current_exception());
}
```