5 Exception safety

Handling errors and exceptions

Preview

- different levels of exception safety
- support mechanisms for exception safety
- implementing exception safety
- exception safety in the standard C++ library

Background

- originally, exceptions and error handling were very poorly understood aspect of C++
  - expection safety issues and requirements were seen and added to the C++ standard at very last moments
- Stroustrup's own C++ book discusses exception safety in an extra Appendix E: *Standard-Library Exception Safety* - (also available from http://www.research.att.com/~bs/)
- two concerns:
  - class invariants must be maintained / restored
  - no resources may be leaked
    - including: memory space, opened files, locks, connections, and any other system resources

Background (cont.)

- exception safety is essential for reusable units (libraries)
- the C++ standard library provides at least the basic exception guarantee: invariants are maintained, and no resources are leaked
- of course, the same should hold for all reusable library components (own or third-party)

Note

- C++ standard libraries still do not necessarily check for all errors (following C-style error handling strategy)
- but when they check - and potentially throw exceptions - they *do* guarantee exception safety
- for many simple (say, only a single added element) operations, the C++ standard library guarantees *strong exception safety* (discussed later)

Revision on invariants

- use defensive programming and self-checking objects
- a *class invariant* is an assertion that holds before and after any operation manipulating an object
- preconditions tests external failures which the unit cannot handle itself: must throw an exception

```
if (!precondition || other external failure)
   throw AnException ("diagnostics" );  // to the caller
```

- invariants and (concrete) postconditions test internal states that don't make sense to outsiders, and may indicate a bug in code => use asserts to eliminate them

```
assert (isInValidInternalState_); // aborts if not
```

- often, we don't know the original reason of a failure: perhaps a programming error or some external factor

Revision on invariants (cont.)

- preconditions and external failures provide a pragmatic trade-off what to check, at the boundary of a unit
- note that the C++ standard library uses the same strategy (checks for selected operations and may throw)

The *fundamental problem* with exception safety

- an exception thrown from some component or function may interrupt the algorithm and leave the state of the calculation (objects) in some indeterminate state
  - the class invariant does not hold => the object cannot be even *destructed* without causing undefined behavior

Many sources of exceptions

- user-supplied and system functions, such as allocator functions, can throw exceptions (from [Stroustrup])

```
void fun (std::vector <X>& v, X const& x) {
    v [2] = x;                    // X's assignment may throw
    v.push_back (x);  // vector <X>'s allocator may throw
    std::sort (v.begin (), v.end ());  // less-than may throw
    std::vector <X> u = v;        // X's copy ctor may throw
    . . .
}   // u is destructed here:   X's dtor should not throw!
```


Levels of exception safety

1. *Basic Guarantee*: no leaks, and maintains invariants.
2. *Strong Guarantee*: succeeds, or leaves state unchanged.
3. *Nofail Guarantee*: doesn't fail in any circumstances.

- the last one (*Nofail*) is often needed to implement the former ones; e.g., an assignment of a primitive value (pointer) cannot fail
- the *strong guarantee* for a complicated update may require a "roll-back" mechanism (can be too expensive)
- "*maintaining invariants*" means
  - the object is in *some* valid state (but not necessarily in the one we would like it to be)
  - but it can be at least released and *destructed*

Exception safety (cont.)

- e.g., std::vector<T>::push_back is designed to give the *strong guarantee*: the item is added or no change

Additionally:

4. *Exception Neutrality*: exceptions originating from components are always passed through unmodified
   - relevant for a container handling and copying its elements, especially when using C++ templates
   - e.g., standard vector<T>::push_back also manifests exception neutrality: after any internal clean-up, propagates the original exception caused by the copying operation - that depends on the actual element type (T)

Notes on exception safety levels

- exceptions are necessary for making reusable libraries and components work: a component may detect an error (violation of invariant) but doesn't know how to handle it
- exception safety means the capability to handle a throw caused by a failure, especially to manage resources

No exception safety

- a failed operation may leave an object in an invalid state (breaking class invariant) and/or leak resources
- this strategy may well be OK: just exit and let the user run the program again

Basic guarantee

- maintain class invariants and don't leak resources
- the processing might possibly be resumed; *at least* the object can be destructed when propagating the exception

Levels of exception safety (cont.)

Strong guarantee

- either an operation succeeds or it doesn't cause any changes (IO operations should behave in a similar way: either succeed, or leave the variable untouched)

Nofail guarantee (also called: "*Nothrow guarantee*")

- an operation cannot ever fail (and throw); e.g., assigment of a primitive value (say, a pointer) cannot fail; the STL container functions size or swap do not throw
- especially, destructors must provide the nofail guarantee

The basic strategy for exception safety

- first calculate results separately; this may succeed or fail (and throw)
- if calculation succeeds, only then make changes in a safe way (that cannot cause any throw)

Calling constructors and destructor

- constructors & destructors are usually called by compiler

```
X * ptr = new X;    // reserve memory, then construct X
delete ptr;         // destruct X, then release memory
```

  – note that if p is zero (0), *delete* has no effect

- when necessary, *allocation* can be separated from object *initialization* with the so-called *placement-new* operator

```
void * p = ::operator new (sizeof(X));  // allocate space
ptr = new (p) X;     //  placement-new constructs X at p
```

- similarly, we can can separate destruction & deallocation

```
ptr->~X ();              // destruct the object pointed by ptr
::operator delete (ptr);    // delete operator frees space
```

  – since destructor is a member function, you can call it explicitly - but then must ensure that compiler doesn't!

Language support for exception safety

- C++ language rules ensure that exceptions thrown while construction will be handled correctly
  - either the object is *fully built* (its invariants OK), or its members become (automatically) destructed
- also *new* operations are implemented safely; "p= new T;" is compiled into something like:

```
p = ::operator new (sizeof (T));      // may fail & throw
try {
    new (p) T;        // placement-new: create a T here
} catch (...) {       // note the exception neutrality
    ::operator delete (p);     // release raw memory
    throw;     // rethrow (dtor is not called, of course)
}
```

- of course, T::T () is assumed to be "safe": has no leaks

On implementing strong guarantee

- sample of strong exception guarantee and roll-back

```
void doOperation (T const& someValue) {
    try {
        <update the state copying the giving value>;
          // e.g., the copy operation may fail & throw
    } catch (...) {            // catch any exception
        <restore the old state and its invariants>;
        throw;            // now rethrow the original
    }
}
```

- exception neutrality: T-related exceptions pass through
- strong guarantee may be very tricky or too costly to achieve; STL does not provide it for all its operations
- special C++ idioms support strong guarantee (see later)

Example: exception safe constructor

```
Vector::Vector (size_t sz, T const& x) {         // illustrative
    rep_= (T* )::operator new (sz*sizeof(T)); // new may fail
    T * p = rep_;                          // element address
    try {                                // construct sz items
       for (; p != rep_+ sz; ++p)  new (p) T (x); // ctor may fail
    } catch (...) {              // handle T constructor failures
       while (p-- != rep_)     // destroy all constructed items
          p->T::~T ();                    // call T's destructor
       ::operator delete (rep_);             // release memory
       throw;           // propagate the original exception
    }
    size_= capacity_= sz;             // OK: Vector initialized
}
```


Exception safety (cont.)

- similar implementation for copy construction: if copy of an item fails, must destruct previously copied ones
- assignment operators can often be safely programmed with an existing copy constructor and swap

```
X& X::operator = (X const& rhs) {
    X tmp (rhs);             // may fail
    swap (tmp);              // does not fail
    return *this;
}
```

- here we trust that the swap operation does not throw
- the same requirement for X's destructor (tmp becomes destroyed at the end of the function before the return)

Destructors are critical for exception handling

Question:

What happens if an exception is thrown out from a destructor while the system is still propagating another?

- destructors should not (generally) allow exceptions to escape from them
  - since propagating an exception calls destructors and if such a destructor lets its exception escape, the program is immediately terminated by system
- so a destructor should trap all local exceptions
  - handle and recover from the exception, or log out an error diagnostics and shut down the program
- the compiler cannot check, and so it is the programmer's responsibility

Example: destructors "throwing exceptions"

```
struct B {  // a class with bad behavior
   ~B () {  // doesn't trap local exceptions
     bool b = std::uncaught_exception (); // for tracing
     throw string ("~B error");  // lets exception escape
   }
};

struct A {   // has a "complicated" destructor that
   ~A () {   // uses a B with its bad behavior
     try {  B b;                          // ~B throws but
     } catch (std::string const& s) { // exception trapped
         assert (s == "~B error"); } // "handled" internally
   }
};
```

```
int main () {
    try {  B b;                          // ~B throws "~B error"
    } catch (string const& e) {          // is caught
        assert (e == "~B error");        // matches ok
    }  // throw out from ~B is handled ok
    try {  A a;
        throw string ("A error");        // ~A handles ~B throw
    } catch (string const& e) {          // original is caught
        assert (e == "A error");         // and matches ok
    }  // local ~B exception inside ~A is handled OK
    try {  B b;
        throw string ("error");          // ~B throws, too
      // => program stopped since ~B exception escapes
    } catch (...) {  /* never comes here  */  }
}
```


Case: how safety mechanisms work

- consider the following C++ class and code

```
class A : public B {
public:
    A () { }          // implicit ctor calls
    X x;   Y y;       // two public members
};   . . .            // implicit dtor
A * a = new A;   . . .
delete a;
```

- the A constructor may seem empty but actually it handles the construction of B, X, and Y parts of an A object
- similarly A's compiler-generated destructor handles the destruction of all these members


```
struct A_Impl  {   // class A's hypothetical implementation
   // reserve space for the data members of A
   char b [sizeof (B)]; char x [sizeof (X)]; char y [sizeof (Y)];
   A_Impl  () {
      new (&b) B;                    // call B::B () may throw
      try { new (&x) X; }            // call X::X () may throw
      catch (...) {
         ((B*)&b)->~B ();  throw;    // destruct B part & rethrow
      }
      try { new (&y) Y; }            // call Y::Y () may throw
      catch (...) {                  // destruct B part & member
         ((X*)&x)->~X ();  ((B*)&b)->~B ();  throw;
      }  // otherwise: an A is now constructed OK
   }
   ~A_Impl  () {      // destruct all its members, in reverse order
      ((Y*)&y)->~Y ();  ((X*)&x)->~X ();  ((B*)&b)->~B (); }
};
```


Case (cont.)

```
A * p = new A;   . . .         // create a dynamic A and use it
delete p;                      // later get rid of it
```

- using the class A_Impl, above code is implemented as

```
void * p = ::operator new(sizeof (A_Impl));  // (1) allocate
        // operator new throws std::bad_alloc upon failure
try {
    new (p) A_Impl; }          // (2) create an A at p (or fail)
catch (...) {
    ::operator delete (p);
}
 . . . // some other code
((A_Impl*)p)->~A_Impl ();      // (1) release A resources
::operator delete (p);         // (2) release p's space
```


Need for exception safety

Reusable library components vs. basic applications

- different levels of exception safety can be identified and are appropriate in different situations
- strong guarantee may be too expensive or not worth it: not all processing or programs can be made or need to be "failure safe"

For example

- an application program is not necessarily meant to be a separate reusable component (or a part of a library)
- when encountering an error, a simple application program may report errors, decide to end its execution, discard all calculated results, and require the user to try it again with more valid input

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*)	—

From Appendix E [Stroustrup]

Exceptions and ctors/dtors: summary

The following built-in C++ mechanisms enable resource management even in case of failures and exceptions

1. an exception throw causes the unwinding of call stack
   – all objects located between the places where the exception is thrown and caught are destroyed, i.e., their destructors are called
2. suppose that an exception is thrown inside a constructor, which has already constructed one or more members
   – the run-time system calls the destructors of the already constructed members to release resources reserved by those members
3. a failed "new X" operation always
   – releases the space allocated for the X object

Summary

- write exception-safe class libraries and components
- three different levels of exception safety can be provided: ***basic***, ***strong***, and ***nofail***
- ***exception neutrality*** needed especially for templates (unknown type parameters with unknown exceptions)
- play safe to prevent bugs and to debug
  – make redundant checks to verify assumptions
  – always initialize everything (especially pointers) to minimize random and unpredictable states
  – remember to clean up resources
  – for raw pointers use ***smart pointers*** (discussed later)