Exceptions

David Rabinowitz

The Role of Exceptions

- Definition: a method *succeeds* if it terminates in a state satisfying its contract. It *fails* if it does not succeed.
- Definition: An *exception* is a runtime event that may cause a routine to fail.
- Exception cases
  - An assertion violation (pre-, post-, invariant, loop)
  - A hardware or operating system problem
  - Intentional call to *throw*
  - A failure in a method causes an exception in its caller

Disciplined Exception Handling

- Mistake 1: Handler doesn't restore stable state
- Mistake 2: Handler silently fails its own contract
- There are two correct approaches
  - Resumption: Change conditions, and retry method
  - Termination: Clean up and fail (re-throw exception)
- Correctness of a *catch* clause
  - Resumption: *{ True } Catch { Inv ∧ Pre }*
  - Termination: *{ True } Catch { Inv }*

Improper Flow of Control

- Mistake 3: Using exceptions for control flow

```
try { value = hashtable.find(key); }
catch ( NotFoundException e ) { value = null; }
```

- It's bad design
  - The contract should never include exceptions
- It's extremely inefficient
  - Global per-class data is initialized and stored
  - Each *try*, *catch*, or exception specification cost time
  - Throwing an exception is *orders of magnitude slower* than returning from a function call

Constructors & Destructors

- Never let an exception leave a destructor
  - In C++: Throwing an exception while destroying due to another exception causes *terminate()*
  - In *finalize()*: The *finalize()* method is stopped
  - The result is resource leaks (yes, in Java too)
- C++ doesn't destroy partially built objects
  - Throwing an exception in a constructor after resources were allocated leads to resource leaks
  - Either split initialization to a method, or avoid pointers
  - Use auto_ptr<T> members instead of T* pointers for const members or members initialized in constructor

Case Study: Genericity

- It's *very* difficult to write generic, reusable classes that handle exceptions well
  - Genericity requires considering exceptions from the template parameters as well
  - Both default and copy constructors may throw
  - Assignment and equality operators may throw
  - In Java: constructors, *equals()* and *clone()* may throw
- See Tom Cargill paper's Stack<T> class code
- "Warm-up" bugs not related to exceptions:
  - Copy, assignment do not set *top* in empty stacks
  - Assignment does not protect against self-assignment

Goals

- Exception Neutrality
  - Exceptions raised from inner code (called functions or class T) are propagated well
- Weak Exception Safety
  - Exceptions (either from class itself or from inner code) do not cause resource leaks
- Strong Exception Safety
  - If a method terminates due to an exception, the object's state remains unchanged

Case Study: Restoring State

- Bug: Throwing OutOfMemory in *push()*
  - *top* has already been incremented
  - *count(), push(), pop()* can no longer be used
  - Fix: restore *top* when throwing the exception
- Bug: Throwing OutOfMemoty in *operator=()*
  - Old array is freed before new one allocated
  - In x=y, x would be inconsistent after failure
  - Fix: Allocate new array into a temporary first

Case Study: Memory Leaks

- Bug: What if *T.operator=()* throws?
  - It can happen: *stack<stack<int>>*
  - See assignment in for loop of copy constructor
  - If T throws here, no stack destructor is called
  - The array allocated in the first line is leaked
- Bug: Same problem in *push()*
  - Again, assignment in for loop may throw
  - Only *new_buffer* points to allocated array
  - In case of exception, this array will be leaked

Case Study: More on State

- Bug: *pop()* ends with *return v[top--];*
  - This involves copying the returned object
  - What if its copy constructor throws?
  - *top* has already been decremented
  - State of the stack has changed, and the client cannot retry to pop the same element

Case Study: More on Memory

- Bug: *operator=()* assumes *T* constructs well
  - First line: *delete []v;*
  - Second line: *v = new T[nelems = s.nelems];*
  - If T's default constructor throws, then v is undefined
  - This can lead to double delete:

```
{ stack x, y;
  y = x;        // exception thrown – y.v is deleted
}               // end of scope – y.v is re-deleted
```


Case Study: Conclusions

- Paper's title: "a false sense of security"
- Combining exceptions and genericity is extremely difficult – STL handles this
- Java has many of the same problems
  - Constructors, *equals(), hashCode(), clone(), toString()* and other *Object* methods may throw
  - In collections: *add(), addAll(), removeAll(), retainAll(), hashCode(),* etc. may throw
  - The *throws* keyword is no help since without genericity, many classes work with *Object* or with other high-level interfaces

Guidelines for Exceptions

- When propagating an exception, try to leave the object in the same state it had in method entry
  - Make sure const functions are really const
  - Perform exception-prone actions early
  - Perform them through temporaries
  - Watch for side effects in expression that might throw
- If you can't leave the object in the same state, try to leave it in a stable state
  - Either re-initialize it, or mark it internally as invalid
  - Do not leave dangling pointers in the object – delete pointers and free resources through temporaries

Guidelines for Exceptions II

- Avoid resource leaks
  - In constructors, initialize raw pointers or resource handles to null first and initialize them inside a try..catch block in the constructor body
  - Don't throw exceptions from a destructor / finalize()
- Don't catch any exceptions you don't have to
  - Rewrite functions to preserve state if possible

    push() {v_[top_] = element; top_++; }
  - Use *catch(…)* to deal with propagating exceptions

    Restore state and re-throw exception

Guidelines for Exceptions III

- Don't hide exceptions from your clients
  - Always re-throw an exception caught with *catch(…)*
  - Throw a different exception only to add information
  - Make sure one *catch* block doesn't hide another
- Use exception hierarchies
  - Define base exception classes for each library
  - Don't be afraid of a deep hierarchy
  - Consider inheriting a standard exception
- Don't get too paranoid

Summary

- Software Correctness & Fault Tolerance
- Design by Contract
  - When is a class correct?
  - Speed, Testing, Reliability, Documentation, Reusability, Improving Prog. Languages
- Exceptions
  - What happens when the contract is broken?
  - Neutrality, Weak Safety, Strong Safety