Understanding C++ Exceptions

Chuck Allison
Utah Valley State College
C/C++ Users Journal
www.freshsources.com

FRESH SOURCES

Agenda

- The Philosophy of Exceptions
- The Mechanics of Exceptions
- Exceptions and Resource Management
- Exception Specifications
- Exception Safety
- (Based in part on material from *Thinking in C++, Volume 2*, by Eckel & Allison)

Pop Quiz!

- What does printf( ) return?

Leading Question

- When was the last time you checked the return value from printf( )?

Error Handling via Return Codes

- You don’t always check them
  - (Did I make my point? :-)
- If you do, the extra code clutter obscures the readability of your program logic
- Even if no errors occur, you’re always wasting cycles checking for them
  - applies to other error-code schemes as well
    - e.g., errno

The Philosophy of Exceptions

- You can’t ignore them
  - Handle them or die!
- Error handling code is localized
  - Code is more readable
- Your code runs faster!
  - If no errors occur
- Yes, there is a space penalty
  - But it’s minimal and worth it!

```
// Illustrates handling “deep errors”
#include <iostream>
using namespace std;

void h()
{
    throw "h() has a problem";
}

void g()
{
    h();
    cout << "doing g..." << endl;
}

void f()
{
    g();
    cout << "doing f..." << endl;
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(const char* msg)
    {
        cerr << "Error: " << msg << endl;
    }

    cout << "back in main" << endl;
}

/* Output:
Error: h() has a problem
back in main
*/
```

Preliminary Details

- The purpose of a try-block is to place exception handlers ("catch-clauses") into the execution stream
- The **`throw`** expression transfers control to an upstream handler
  - the nearest-enclosing "matching" handler
    - according to the type of exception thrown
  - so it can recover from the error

Pretty Good Idea #1

- Use exceptions to indicate *errors*
- For functions that can’t fulfill their specification
- Not for alternate returns under normal circumstances

Potential Problem

- What if local objects are created?
  - In f( ), g( ), say
- They may need their destructor called
- Not a problem

Stack Unwinding

- As execution backtracks up the call stack, local objects have their destructors called
- Allows for convenient resource deallocation
  - A key to exception safety

```
#include <iostream>
using namespace std;

void h()
{
    Foo f3;
    throw "h() has a problem";
}

void g()
{
    Foo f2;
    h();
    cout << "doing g..." << endl;
}

void f()
{
    Foo f1;
    g();
    cout << "doing f..." << endl;
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(const char* msg)
    {
        cerr << "Error: " << msg << endl;
    }

    cout << "back in main" << endl;
}

/* Output:
Foo
Foo
Foo
~Foo
~Foo
~Foo
Error: h() has a problem
back in main
*/
```

How to Throw Exceptions

- **`throw`** keyword
- Throw objects of user-defined classes
  - Can hold auxiliary information
  - Allows clear categorization of errors
- Use constructor syntax

```
// Exception class
class MyError
{
    string msg;
public:
    MyError(const string& s) : msg(s) {}
    string what() {return msg;}
};

// ...
```

```
void h()
{
    throw MyError("h() has a problem");
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(MyError& x)
    {
        cerr << "MyError: " << x.what() << endl;
    }

    // Control goes here ("termination semantics")
    cout << "back in main" << endl;
}
```

Catching Exceptions

- Execution backtracks until it finds a matching handler
- Exact type, or
- An accessible base class type
- Beware built-in types
  - rules are complicated; use classes!
  - string literals are **`const char*`**
    - (not caught via a **`char*`** catch parameter)
- Not all conversions apply!
  - Sufficient info not available at runtime!

Exceptions and Conversions

```
class Except1 {};
class Except2 {
public:
  Except2(Except1&) {}
};

void f() { throw Except1(); }

int main() {
  try {
    f();
  } catch (Except2&) {
    cout << "inside catch(Except2)" << endl;
  } catch (Except1&) {
    cout << "inside catch(Except1)" << endl;
  }
}

/* Output:
inside catch(Except1)
*/
```

If **`D`** derives from **`B`**…

- **`catch(B&)`** catches a **`B`** or a **`D`**
  - so order of handlers in code matters!
  - B must be an unambiguous, public base for D
- **`catch(B*)`** catches a **`B*`** or **`D*`**
- **`catch(void*)`** catches all pointer types

Order Matters!

- Handlers are tried in order of their appearance in the code
- Most specific handlers should appear first
- Derived class handlers should precede base class handlers
- catch(…), if present, should be last

Uncaught Exceptions

- If no handler is found, the library function `terminate()` is called
  - Which just calls `abort()`
- If you want to prevent termination:
  - Make sure all exceptions are caught!
- You can install your own *terminate handler*
  - With `set_terminate()`

What should `terminate` do?

- Log the error
- Tidy-up as needed (release global resources, if any)
- exit the program
- `terminate` *cannot:*
  - return
  - throw exceptions

set_terminate

```
#include <iostream>
#include <exception>   // for set_terminate()
#include <cstdlib>     // for exit()
using namespace std;

void handler()
{
    cout << "Renegade exception!\n";
    exit(1);
}

int main()
{
    void f();
    set_terminate(handler);

    try
    {
        f();
    }
```

```
    catch(long)
    {
        cerr << "caught a long" << endl;
    }
}

void f()
{
    throw "oops";   // Doesn't match a long
}

// Output:
Renegade exception!
```

`terminate()` is called when...

- A matching handler is not found, including when:
  - a constructor for a static object throws
  - An *exit handler* (from `atexit`) throws
- A destructor throws during stack unwinding
  - Only one exception at a time, thank you!
  - Destructors shouldn’t emit exceptions

How does all this really work?

- **`throw`** is *conceptually* like a function call
  - Takes the exception object as a “parameter”
- This special “function” backtracks up the program stack (the dynamic call chain)
  - Reading information placed there by *each function invocation*
    - Information placed in each “Stack Frame”
    - About each function’s local objects and try blocks
- If no matching handler is found in a function, local objects’ are destroyed and the search continues
  - Until a matching handler is found
  - Or terminate( ) is ultimately called

Space Overhead

```
struct C
{
    ~C(){}
};

void g();    // for all we know, g may throw

void f()
{
    C c;     // Destructor must be called
    g();
}
```

Compiler Exception Support

- Microsoft Visual C++ .NET (-GX)
  - 1,420 bytes vs. 2,069 bytes
- Borland C++ Builder 6.0 (-x-)
  - 813 bytes vs. 2,150 bytes

Runtime Overhead

- Two Types
  - Adding exception-related info to each stack frame
  - The work done during stack unwinding
    - This is *good overhead*, since you want things cleaned up
    - Following return-code paths the old-fashioned way has a cost too, you know!

The Zero-cost Model

- Adorning each stack frame with exception-related info can have a *runtime* cost
- Can be avoided
  - Offsets for objects with destructors can be computed once at compile time and stored outside the runtime stack
- GNU and Metrowerks compilers currently support this

Another Leading Question

Since exception objects originate in a different scope from where they’re caught, how are they accessible in a handler?

Answer

- Exception objects are *temporaries*
  - A *copy* is thrown
    - Const-ness is stripped away (except for string literals)
  - Exceptions must be *copyable* and *destructible*
    - accessible in the context of the throw expression
- Catching by *value* creates an *additional copy*
  - And derived objects caught as a base are sliced
- Catch-by-pointer, is problematic (how to know whether you have to `delete` it)?

Pretty Good Idea #2

- Catch exceptions by reference.
- What about *const* reference?
  - A local stylistic concern
  - Const and volatile are ignored in finding a matching handler
  - You can modify the exception object as it moves up the stack
    - because the same object is re-thrown

Standard Exceptions

- Thrown by the Standard Library
- Hierarchy of *Logic* vs. *Runtime* Errors
- **`exception`** base class

Standard Exceptions

- **`exception`**
  - **`logic_error`** (client program error)
    - `domain_error`, `invalid_argument`, `length_error`, `out_of_range`
  - **`runtime_error`** (external error)
    - `range_error`, `overflow_error`, `underflow_error`
  - `bad_alloc` (memory failure)
  - `bad_cast` (bad `dynamic_cast` w/ref)
  - `bad_exception` (`unexpected`)
  - `bad_typeid` (`typeid` w/null)

```
try
{
    string s;
    cout << s.at(100) << endl;  // invalid arg
}
catch (logic_error& x)
{
    cout << "logic_error: " << x.what()
         << endl;
}
catch (runtime_error& x)
{
    cout << "runtime_error: " << x.what()
         << endl;
}
catch (exception& x)
{
    cout << "exception: " << x.what()
         << endl;
}
```

```
// Output:
logic_error: position beyond end of string
```

Using Standard Exceptions

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

// Exception class (polymorphic because
// std::exception is)
struct MyError : runtime_error
{
   MyError(const string& msg)
   : runtime_error(msg){}
};
```

```cpp
int main()
{
    try
    {
        f();
    }
    catch (MyError& x)
    {
        cerr << x.what() << endl;
    }
    catch (exception& x)
    {
        cerr << x.what() << endl;
    }
    catch (...) // catch-all
    {
        cerr << "Unknown error\n";
    }

    cout << "back in main" << endl;
}
```

```
// Using RTTI (a sometimes-useful trick):
int main()
{
    try
    {
        f();
    }
    catch(exception& x)
    {
        cerr << typeid(x).name() << ':'
             << x.what() << endl;
    }
    catch (...)     // catch-all
    {
        cerr << "Unknown error\n";
    }

    cout << "back in main" << endl;
}
```

```
MyError:h()has a problem
Back in main
```

Pretty Good Idea #3

- Throw objects of classes derived (ultimately, not necessarily directly) from **`std::exception`**
- (**`std::exception`** does not take a std::string parameter in its ctor)

Exceptions and IOStreams

- How do you test for stream errors?
  - `if (strm.fail())`…   `if (!strm)`…
- You're checking a "return value"!
- You can have stream errors throw:
  `strm.exceptions(ios::failbit);`
- An `ios::failure` exception is thrown

What Should a Handler Do?

- Fully recover, then resume somehow, or
- Partially recover and *re-throw* the exception

  (by using **`throw;`**)

Pretty Good Idea #4

- If you can’t do anything about an exception, don’t catch it!
- Unless you need to release resources
  - then re-throw the exception

Pretty Good Idea #5

- `catch(...)` should usually re-throw

Resource Management

- Dangling Resource Problem
  - a function that allocates a resource might throw before deallocating the resource
- Solutions:
  - Handle the situation locally
  - use an Object Wrapper (RAII)
- **`auto_ptr`**, the standard wrapper for memory
  - a *smart pointer*

A Dangling Resource

```
void f(const char* fname)
{
    FILE* fp = fopen(fname,"r");
    if (fp)
    {
        g(fp);         // Suppose g() throws?
        fclose(fp); // Then this won't happen!
    }
}

// continued...
```

Local Handlers

```
void f(const char* fname)
{
    FILE* fp = fopen(fname,"r");
    if (fp)
    {
        try
        {
            g(fp);
        }
        catch(...)
        {
            fclose(fp);
            puts("File closed");
            throw;  // Re-throw for
                    // other handlers
        }
        fclose(fp); // The normal close
    }
}
```

RAII

- “Resource Allocation is Initialization”
- Use objects on the stack to control resources
- The constructor allocates
- The destructor deallocates

Object Wrappers

(To leverage stack unwinding)

```
class File
{
    FILE* f;

public:
    File(const char* fname, const char* mode)
    {
        f = fopen(fname, mode);  // allocate
    }
    ~File()
    {
        fclose(f);              // deallocate
        puts("File closed");
    }
};
```

```
void f(const char* fname)
{
    File x(fname,"r");
    g(x.getFP());
}
```

Pretty Good Idea #6

- Use object wrappers to manage resources

Memory Leaks

```
void f()
{
    T* p = new T;
    g(p);       // Suppose g() throws?
    delete p;   // Then this won't happen!
}
```

auto_ptr

```
void f()
{
    auto_ptr<T> p(new T);
    g(p);
}

// delete p is implicit
```

Another **auto_ptr** Example

```
Employee* Employee::read(istream& in)
{
  // Create object from file data
  auto_ptr<Employee> p(new Employee);
  in >> *p;
  if (in.fail())
    throw EmployeeError("File input error");
  return p.release();
}
```

Pretty Good Idea #7

- Wrap local & member heap allocations in an `auto_ptr` object
  - scalars only – no arrays!
- Don’t do much else with it
  - Herb Sutter, “Using `auto_ptr` Effectively”, CUJ, October 1999, pp. 63-67.

Dynamic Memory Mgt.

- **`new`** operator throws **`bad_alloc`** when memory is exhausted
- You can request traditional null-return behavior with **`nothrow_t`** version
- Or call **`set_new_handler`** to install your own new handler

`new` and Exceptions

```
#include <new>
#include <iostream>

int main()
{
    try
    {
        int* p = new int;
        cout << "memory allocated\n";
    }
    catch (bad_alloc& x)
    {
        cout << "memory failure: " << x.what()
             << endl;
    }
}
```

new - Traditional Behavior

```
#include <new>
#include <iostream>
using namespace std;

int main()
{
      int* p = new (nothrow) int;
      if (p)
            cout << "memory allocated\n";
      else
            cout << "memory failure\n";
}
```

Exception Specifications

- To control what exceptions are thrown
- Not just documentation
- An enforced *specification*
  - enforced at *runtime*
  - Except non-covariant, derived ES's are caught at *compile time*
- Controversial commentary:
  - ES's are not widely used

Exception Specifications

```
class A;
class B;

void f() throw(A,B)
{
    // Whatever
    g();
}
```

Can also throw objects derived from `A` or `B`.

```
// Equivalent to:
void f()
{
    try
    {
        // Whatever
        g();
    }
    catch(A&)
    {
        throw;                // rethrow
    }
    catch(B&)
    {
        throw;                // rethrow
    }
    catch(...)
    {
        std::unexpected();
    }
}
```

Exception Specifications

- `void f() throw()`
  - No exceptions allowed
- `void f()`
  - Can throw any exception
- Not part of the type of a function
  - can’t use with **typedef** or overloading

Unexpected Handlers

- The default `unexpected()` calls `terminate()`
- You can replace it via `set_unexpected()`

(20-25 minutes left – skip 6)

What should `unexpected()` do?

- Log the error
- Abort
  - You need to fix your program!
- `unexpected` cannot return
  - but it can throw (see next slide)

Mapping Exceptions

- You can leave some exceptions unspecified and catch them in one place
    - Using an **`unexpected`** handler and **`bad_exception`**
- A “work-around” for not having unchecked exceptions like Java does

bad_exception

- A special way to map unexpected exceptions to a single type
- Just add **`bad_exception`** to the specification:

```
void f() throw (std::bad_exception)
```

- You must install an **`unexpected()`** that throws
- The original exception is lost

```
// Works on GNU
#include <exception>
#include <iostream>
using namespace std;

void handler()
{
    cerr << "unexpected exception\n";
    throw;
}

void g()
{
   throw 1;
}

void f() throw(bad_exception)
{
    g();
}
```

```
int main()
{
    set_unexpected(handler);
    try
    {
        f();
    }
    catch (bad_exception&)
    {
        cout << "caught exception\n";
    }
}
```

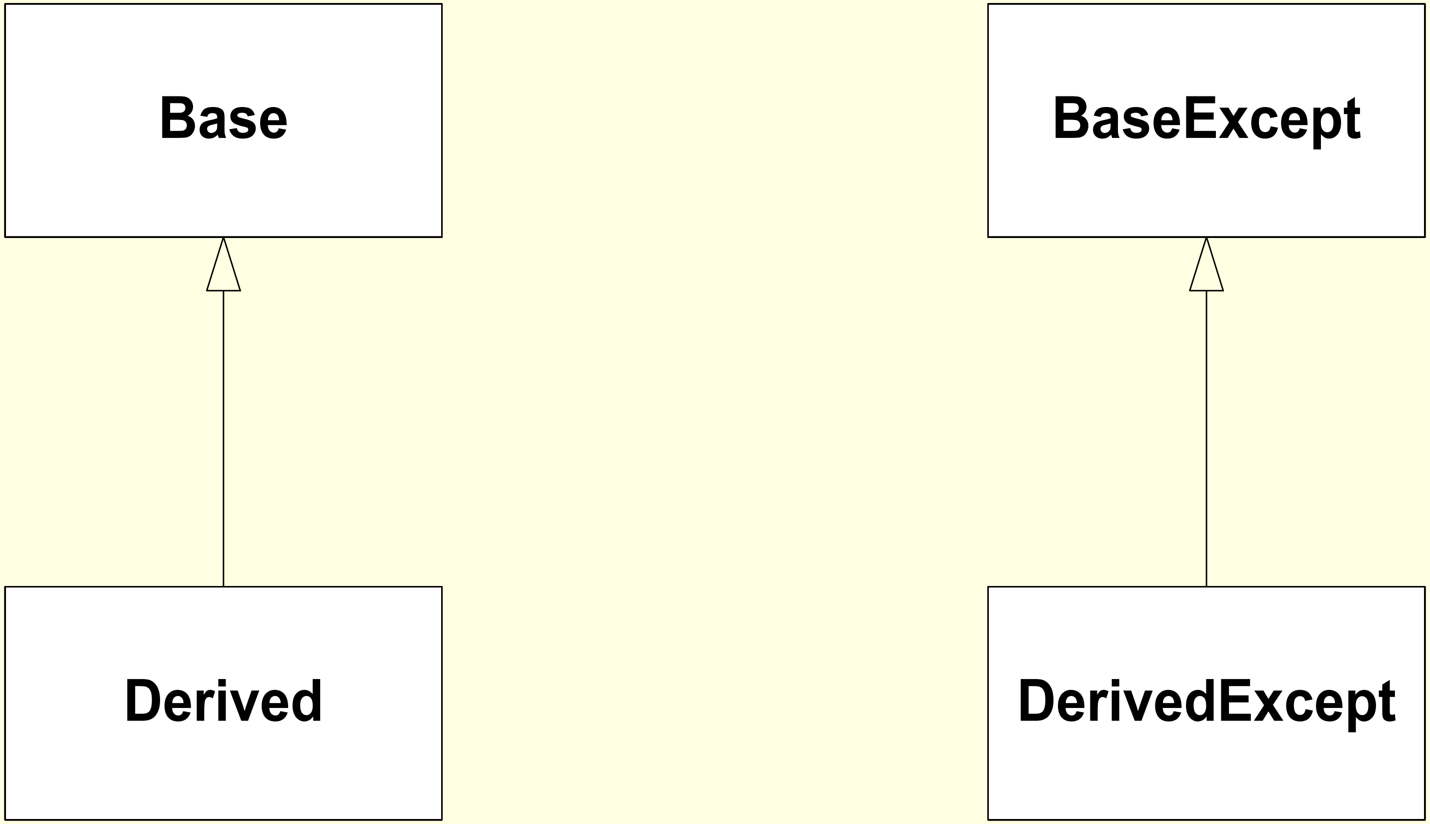
```
unexpected exception
caught exception
```

Managing Unexpected Exceptions

- Can wrap calls to **`set_unexpected`** in a class:
  - constructor sets **`unexpected`** to throw a *user-defined* exception
  - destructor resets **`unexpected`**
- See Stroustrup, 3rd Edition, pp. 378-380

Exception Specifications and Inheritance

- Functions in derived classes must not *expand* the exception specification list of the base class function they override
- This would break the base class' contract
- But they can throw exceptions *derived* from those in the base class method's list
  - Fancy term: *covariance*
- They can also specify *fewer* exceptions
  - Because the contract is still preserved
- Example: next two slides



Base

Derived

BaseExcept

DerivedExcept

```
class Base
{
public:
   virtual void f() throw(BaseExcept);
};

class Derived : public Base
{
public:
   // Any of these three is okay:
//    void f() throw(BaseExcept);
//    void f() throw(DerivedExcept);
//    void f() throw() {}

   // These would be errors (caught at compile time):
//    void f() throw(RogueExcept);
//    void f() {}
};
```

What if **`g()`** throws?

```
void f() throw(A,B)
{
    // Whatever…, then:
    g();
}
```

Pretty Good Idea #8

- If `f()` calls `g()`, and `g()` has no exception specification, don't declare an exception specification for `f()`

Exception Specifications and Templates

- They just don’t mix!
- You never know what a generic type might do
    - Containers call copy constructors and assignment operators a lot
        - Which can throw exceptions
- Especially crucial with container design
- You can use throw( ), of course

Rule of Generic Container Design

- Don’t use Exception Specifications with generic containers
- Instead, document the exception you know about
  - This is what the Standard Library does
  - It only uses throw( )

What’s Wrong Here?

```
void StackOfInt::grow()
{
   // Enlarge stack’s data store
   capacity += INCREMENT;
   int* newData = new int[capacity];
   for (size_t i = 0; i < count; ++i)
      newData[i] = data[i];
   delete [] data;
   data = newData;
}
```

An Improvement

```
void StackOfInt::grow()
{
   // Enlarge stack's data store
   size_t newCapacity = capacity + INCREMENT;
   int* newData = new int[newCapacity];
   for (size_t i = 0; i < count; ++i)
      newData[i] = data[i];

   // Update state only when "safe" to do so
   delete [] data;
   data = newData;
   capacity = newCapacity; // moved
}
```

Fundamental Principle of Exception Safety

- Separate operations that may throw from those that change state
  - only change state when exceptions can no longer occur
- Corollary:
  - Do one thing at a time (cohesion)
  - why std::stack<T>::pop() returns void
    - The returned copy might throw
      - and the state has changed!

Rules of Exception Safety

- If you can’t handle an exception, let it propagate up (“Exception neutral”)
- Leave your data in a *consistent* state
  - Use RAII to allocate resources
  - Only change your state with non-throwing ops
  - An object should only own one resource
- Functions should perform only one logical operation
- Destructors should never throw
- Good references:
  - Sutter, *Exceptional C++* and *More Exceptional C++*
  - Abrahams, www.boost.org/more/generic_exeption_safety.html

Levels of Exception Safety

(David Abrahams)

- Basic Guarantee
  - No resources will be leaked
  - Good, but not always sufficient
    - State may be “consistent”, but not “acceptable”
- Strong Guarantee
  - No changes will occur if an exception happens
  - Requires roll-back semantics (not always possible)
    - Iterators may be invalidated, for example
- No-throw Guarantee
  - No exceptions will escape
  - Required of destructors and swap( )

A Safe operator=( )

- First provide a **swap** member function that doesn't throw
  - Just swap state (ptrs and ints) with std::swap
- Then op= swaps its rhs *value* parameter
  - No need to worry about self-assignment (value parm)

```
template<class T>
Stack<T>& Stack::operator=(Stack<T> rhs)
{
    swap(rhs);    // Stack<T>::swap(Stack<T>&)
    return *this;
}
```

Really Good Idea #8

- Don’t let an exception escape from a destructor.
- If you see no alternative, however, make sure an exception isn’t pending with the `uncaught_exception()` library function, then proceed.
  - I’ve never seen it done

```
#include <exception>
#include <iostream>
using namespace std;

class C
{
public:
    ~C()
    {
        if (uncaught_exception())
            cout << "unwinding..\n";
        else
            throw 1;
    }
};
```

```
int main()
{
    try
    {
        C c;
    }
    catch (int&)
    {
        cout << "caught an int\n";
    }
```

```
caught an int
```

```
    try
    {
        C c;
        throw "";
    }
    catch (char*)
    {
        cout << "caught a char*\n";
    }
}
```

```
unwinding..
caught a char*
```

Destructors that Throw

- Are Evil
- Unfit for use in containers
- So use uncaught_exception( ) only under controlled (non-container) conditions

A Bit of Esoterica

- Function-level try blocks
- Rarely used…

Function-level Try Blocks

(as if we care)

```
try
void f(int a, float b)
{
    …
}
catch (T& t)
{
 // a and b in scope here (not f’s locals)
 // can throw or return from here
}
```

Member Initializers

(to care is rare)

- Special syntax to catch member constructors that throw:

```
X::X(Y init)  // suppose X has a y-member
try
     : y(init)
{
     [normal constructor body here]
}
catch (YException& ex) {/* should throw;
  can’t return */}
```

About Object Construction

- An object doesn’t exist until its constructor exits successfully
- If an exception occurs during construction, there is no complete object
- Therefore, in a *constructor handler*, you can’t reliably access an object’s state
- The only thing to do is to throw another exception
    - After writing to log file, say
    - Or you could just let the original exception propagate
    - If you don’t throw, a rethrow is *implicit*

Finis, El Fin, O Fim, The End