

Exception-style resource management

Pre-exception-handling style:

```
void useFile( const char* fn ) {
    FILE* f = fopen( fn, "w" );
    // use f
    fclose( f );
}
```

If "use f" throws an exception, f never gets closed.

A fix:

```
void useFile( const char* fn ) {
    FILE* f = fopen( fn, "w" );
    try {
        // use f
    }
    catch( ... ) {
        fclose( f );
        throw;
    }
    fclose( f );
}
```

It gets worse if a function acquires and releases multiple resources:

```
void useResources() {
    // acquire resource 0
    // acquire resource 1
    // ...
    // acquire resource n
    // use resources
    // release resource n
    // ...
    // release resource 1
    // release resource 0
}
```

Resources typically need to be released in LIFO order, in case resource k depends on resource $k-1$.If acquiring resource k triggers an exception, then only resources $0..k-1$ should be released.

The catch-and-rethrow approach makes a mess of the code:

```
void useResources() {
    // acquire resource 0
    try {
        // acquire resource 1
        try {
            // acquire resource 2
            try {
                // ... etc. for resources [3 ... n]
                try {
                    // use resources
                }
                catch ( ... ) {
                    // release resource n
                    throw;
                }
            }
            // ... etc. for resources [n-1 ... 3]
        }
        catch ( ... ) {
            // release resource 2
            throw;
        }
    }
    catch ( ... ) {
        // release resource 1
        throw;
    }
}
catch ( ... ) {
    // release resource 0
    throw;
}
// release resource n
// ... etc. for resources [n-1 ... 2]
// release resource 1
// release resource 0
}
```

A solution which "scales": Embed *all* resource handles in classes

- acquisition: constructor call
- release: destructor call

Example:

```
class FilePtr {
public:
    FilePtr( const char* n, const char* a )
        : p( fopen(n,a) ) {}
    FilePtr( FILE* pp ) : p(pp) {}
    ~FilePtr(){ fclose(p); }
    operator FILE*(){ return p; }
private:
    FILE* p;
};
```

```
void useFile( const char* fn ) {
    FilePtr f( fn, "w" );
    // use f
}
```

The compiler ensures that local variables' destructors are called in opposite order (LCFD) from their constructors.

Exceptions thrown in destructors

What happens if an exception is thrown during execution of a destructor that is called while an exception is being handled?

- if the exception is handled within the destructor, OK
- if the exception “escapes” from the destructor, `std::terminate()` is called.

Exceptions thrown in constructors

An exception thrown by a member initializer is passed to whatever invoked the member’s constructor. But a constructor can handle its own exceptions:

```
class X {
    Array a;
    // ...
public:
    X( int );
    // ...
};
X::X( int s )
try :a(s) // initialize a with s
{
    // ... constructor body
}
catch( Array::Size ) {
    // exceptions thrown for a are caught here
}
```

26 April 2005

Grouping exceptions

Exception classes can be organized in derivation hierarchies:

```
class MathError{...};
class Overflow : public MathError{...};
class Underflow : public MathError{...};
class ZeroDivisor : public MathError{...};
// ...
```

This enables a catch to handle multiple types of exceptions —provided they share a base class.

```
void f()
{
    try { ... }
    catch ( Overflow& ) // must be first
    { ...handle Overflow and its derivatives... }
    catch ( MathError& )
    { ... all other kinds of MathError ... }
}
```

Grouping means that when new kinds of `MathError` are defined, we don’t have to search for all the places where `MathErrors` are handled.

26 April 2005

Applying this notion to the `Array` class:

```
class ArrayError {};
```

```
class Range : public ArrayError
{ ... }
```

```
class Size : public ArrayError
{ ... }
```

```
try {
    // use Arrays
}
catch( Array::ArrayError& e ) {
    // ... handles both Range and Size
    // exceptions ...
}
```

26 April 2005

Exception specifications

Because a function’s exception-throwing behavior is visible externally, it belongs in the interface specification:

```
int& operator[]( int ) throw (Range);
Array( int ) throw (Size);
```

If a function throws an exception that’s not in its exception list, `std::terminate()` is called (compiler catches only some violations).

All declarations of a function (including its definition) must list the same set of exception types.

A function can guarantee to throw *no* exceptions:

```
void noExcep( int ) throw ();
```

but a function with *no* exception list is allowed to throw *any* exception:

```
void anyExcep( int );
```

This “feature” makes it difficult to write functions with exception lists correctly.

Why are exception lists optional?

- Because exceptions were added to C++ late in the game, long after masses of software had been written (compatibility strikes again!)
- “The point is to ensure that adding an exception somewhere doesn’t force a complete update of related exception specifications and a recompilation of all potentially affected code. ... This is essential for the maintenance of large systems in which major updates are expensive and not all source code is accessible.”
— Stroustrup, *C++, 3e*, p. 376.

26 April 2005