Gotcha #65: Improper Exception Mechanics

Issues of general exception-handling policy and architecture are still subject to debate. However, lower-level guidelines concerning how exceptions should be thrown and caught are both well understood and commonly violated.

When a throw-expression is executed, the runtime exception-handling mechanism copies the exception object to a temporary in a “safe” location. The location of the temporary is highly platform dependent, but the temporary is guaranteed to persist until the exception has been handled. This means that the temporary will be usable until the last catch clause that uses the temporary has completed, even if several different catch clauses are executed for that temporary exception object. This is an important property because, to put it bluntly, when you throw an exception, all hell breaks loose. That temporary is the calm in the eye of the exception-handling maelstrom.

This is why it’s not a good idea to throw a pointer.

```cpp
throw new StackUnderflow( "operator stack" );
```

The address of the StackUnderflow object on the heap is copied to a safe location, but the heap memory to which it refers is unprotected. This approach also leaves open the possibility that the pointer may refer to a location that’s on the runtime stack:

```cpp
StackUnderflow e( "arg stack" );
throw &e;
```

Here, the storage to which the pointer exception object (remember, the pointer is what’s being thrown, not what it points to) is referring to storage that may not exist when the exception is caught. (By the way, when a string literal is thrown, the entire array of characters is copied to the temporary, not just the address of the first character. This information is of little practical use, because we should never throw string literals. See Gotcha #64.) Additionally, a pointer may be null. Who needs this additional complexity? Don’t throw pointers, throw objects:

```cpp
StackUnderflow e( "arg stack" );
throw e;
```

The exception object is immediately copied to a temporary by the exception-handling mechanism, so the declaration of `e` is really not necessary. Conventionally, we throw anonymous temporaries:

```cpp
throw StackUnderflow( "arg stack" );
```
Use of an anonymous temporary clearly states that the StackUnderflow object is for use only as an exception object, since its lifetime is restricted to the throw-expression. While the explicitly declared variable e will also be destroyed when the throw-expression executes, it is in scope, and accessible, until the end of the block containing its declaration. Use of an anonymous temporary also helps to stem some of the more “creative” attempts to handle exceptions:

```cpp
static StackUnderflow e( "arg stack" );
extern StackUnderflow *argstackerr;
argstackerr = &e;
throw e;
```

Here, our clever coder has decided to stash the address of the exception object for use later, probably in some upstream catch clause. Unfortunately, the argstackerr pointer doesn’t refer to the exception object (which is a temporary in an undisclosed location) but to the now destroyed object used to initialize it. Exception-handling code is not the best location for the introduction of obscure bugs. Keep it simple.

What’s the best way to catch an exception object? Not by value:

```cpp
try {
    // . . .
}
catch( ContainerFault fault ) {
    // . . .
}
```

Consider what would happen if this catch clause successfully caught a thrown StackUnderflow object. Slice. Since a StackUnderflow is a ContainerFault, we could initialize fault with the thrown exception object, but we’d slice off all the derived class’s data and behavior. (See Gotcha #30.)

In this particular case, however, we won’t have a slicing problem, because ContainerFault is, as is proper in a base class, abstract (see Gotcha #93). The catch clause is therefore illegal. It’s not possible to catch an exception object, by value, as a ContainerFault.

Catching by value allows us to expose ourselves to even more obscure problems:

```cpp
catch( StackUnderflow fault ) {
```

© 2003 Pearson Education, excerpted from *C++ Gotchas* by Stephen C. Dewhurst
It’s not uncommon for a catch clause to perform a partial recovery, record the state of the recovery in the exception object, and re-throw the exception object for additional processing. Unfortunately, that’s not what’s happening here. This catch clause has performed a partial recovery, recorded the state of the recovery in a local copy of the exception object, and re-thrown the (unchanged) exception object.

For simplicity, and to avoid all these difficulties, we always throw anonymous temporary objects, and we catch them by reference.

Be careful not to reintroduce value copy problems into a handler. This occurs most commonly when a new exception is thrown from a handler rather than a re-throw of the existing exception:

```cpp
catch( ContainerFault &fault ) {
    // do partial recovery . . .
    if( condition )
        throw; // re-throw
    else {
        ContainerFault myFault( fault );
        myFault.modifyState(); // still my fault
        throw myFault; // new exception object
    }
}
```

In this case, the recorded changes will not be lost, but the original type of the exception will be. Suppose the original thrown exception was of type StackUnderflow. When it’s caught as a reference to ContainerFault, the dynamic type of the exception object is still StackUnderflow, so a re-thrown object has the opportunity to be caught subsequently by a StackUnderflow catch clause as well as a ContainerFault clause. However, the new exception object myFault is of type ContainerFault and cannot be caught by a StackUnderflow clause. It’s generally better to re-throw an existing exception object rather than handle the original exception and throw a new one:

```cpp
catch( ContainerFault &fault ) {
    // do partial recovery . . .
    if( !condition )
        fault.modifyState();

    // re-throw current exception
    throw;
}
```
Fortunately, the `ContainerFault` base class is abstract, so this particular manifestation of the error is not possible; in general, base classes should be abstract. Obviously, this advice doesn’t apply if you must throw an entirely different type of exception:

```cpp
catch( ContainerFault &fault ) {
    // do partial recovery . . .
    if( out_of_memory )
        throw bad_alloc(); // throw new exception
    fault.modifyState();
    throw; // re-throw
}
```

Another common problem concerns the ordering of the catch clauses. Because the catch clauses are tested in sequence (like the conditions of an if-elseif, rather than a switch-statement) the types should, in general, be ordered from most specific to most general. For exception types that admit to no ordering, decide on a logical ordering:

```cpp
catch( ContainerFault &fault ) {
    // do partial recovery . . .
    fault.modifyState(); // not my fault
    throw;
}
catch( StackUnderflow &fault ) {
    // . . .
}
catch( exception & ) {
    // . . .
}
```

The handler-sequence above will never catch a `StackUnderflow` exception, because the more general `ContainerFault` exception occurs first in the sequence.

The mechanics of exception handling offer much opportunity for complexity, but it’s not necessary to accept the offer. When throwing and catching exceptions, keep things simple.