Gotcha #64: Throwing String Literals

Many authors of C++ programming texts demonstrate exceptions by throwing character string literals:

```
throw "Stack underflow!";
```

They know this is a reprehensible practice, but they do it anyway, because it's a "pedagogic example." Unfortunately, these authors often neglect to mention to their readers that actually following the implicit advice to imitate the example will spell mayhem and doom.

Never throw exception objects that are string literals. The principle reason is that these exception objects should eventually be caught, and they're caught based on their type, not on their value:

```
try {
   // . . .
}
catch( const char *msg ) {
   string m( msg );
   if( m == "stack underflow" ) // . . .
   else if( m == "connection timeout" ) // . . .
   else if( m == "security violation" ) // . . .
   else throw;
}
```

The practical effect of throwing and catching string literals is that almost no information about the exception is encoded in the type of the exception object. This imprecision requires that a catch clause intercept every such exception and examine its value to see if it applies. Worse, the value comparison is also highly subject to imprecision, and it often breaks under maintenance when the capitalization or formatting of an "error message" is modified. In our example above, we'll never recognize that a stack underflow has occurred.

These comments also apply to exceptions of other predefined and standard types. Throwing integers, floating point numbers, `string`s, or (on a really bad day) `set`s of `vector`s of `float`s will give rise to similar problems. Simply stated, the problem with throwing exception objects of predefined types is that once we've caught one, we don't know what it represents, and therefore how to respond to it. The

thrower of the exception is taunting us: "Something really, really bad happened. Guess what!" And we have no choice but to submit to a contrived guessing game at which we're likely to lose.

An exception type is an abstract data type that represents an exception. The guidelines for its design are no different from those for the design of any abstract data type: identify and name a concept, decide on an abstract set of operations for the concept, and implement it. During implementation, consider initialization, copying, and conversions. Simple. Use of a string literal to represent an exception makes about as much sense as using a string literal as a complex number. Theoretically it might work, but practically it's going to be tedious and buggy.

What abstract concept are we trying to represent when we throw an exception that represents a stack underflow? Oh. Right.

```
class StackUnderflow {};
```

Often, the type of an exception object communicates all the required information about an exception, and it's not uncommon for exception types to dispense with explicitly declared member functions. However, the ability to provide some descriptive text is often handy. Less commonly, other information about the exception may also be recorded in the exception object:

```
class StackUnderflow {
 public:
   StackUnderflow( const char *msg = "stack underflow" );
   virtual ~StackUnderflow();
   virtual const char *what() const;
   // . . .
};
```

If provided, the function that returns the descriptive text should be a virtual member function named `what`, with the above signature. This is for orthogonality with the standard exception types, all of which provide such a function. In fact, it's often a good idea to derive an exception type from one of the standard exception types:

```
class StackUnderflow : public std::runtime_error {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
};
```

This allows the exception to be caught either as a `StackUnderflow`, as a more general `runtime_error`, or as a very general standard `exception` (`runtime_error`'s public base class). It's also often a good idea to provide a more general, but nonstandard, exception type. Typically, such a type would serve as a base class for all exception types that may be thrown from a particular module or library:

```
class ContainerFault {
 public:
   virtual ~ContainerFault();
   virtual const char *what() const = 0;
   // . . .
};
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
   const char *what() const
       { return std::runtime_error::what(); }
};
```

Finally, it's also necessary to provide proper copy and destruction semantics for exception types. In particular, the throwing of an exception implies that it must be legal to copy construct objects of the exception type, since this is what the runtime exception mechanism does when an exception is thrown (see Gotcha #65), and the copied exception must be destroyed after it has been handled. Often, we can allow the compiler to write these operations for us (see Gotcha #49):

```
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
   // StackUnderflow( const StackUnderflow & );
   // StackUnderflow &operator =( const StackUnderflow & );
   const char *what() const
       { return std::runtime_error::what(); }
};
```

Now, users of our stack type can choose to detect a stack underflow as a `StackUnderflow` (they know they're using our stack type and are keeping close watch), as a more general `ContainerFault` (they know they're using our container library and are on the qui vive for any container error), as a `runtime_error` (they know nothing about our container library but want to handle any sort of standard runtime error), or as an `exception` (they're prepared to handle any standard exception).