This is a slightly modified version of an article that appeared in the November-December 1994 issue of the *C++ Report*.

Exception Handling: A False Sense of Security

by Tom Cargill

I suspect that most members of the C++ community vastly underestimate the skills needed to program with exceptions and therefore underestimate the true costs of their use. The popular belief is that exceptions provide a straightforward mechanism for adding reliable error handling to our programs. On the contrary, I see exceptions as a mechanism that may cause more ills than it cures. Without extraordinary care, the addition of exceptions to most software is likely to diminish overall reliability and impede the software development process.

This "extraordinary care" demanded by exceptions originates in the subtle interactions among language features that can arise in exception handling. Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches. The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way.

In the October 1993 issue of the *C++ Report*, David Reed argues in favor of exceptions saying that: "Robust reusable types require a robust error handling mechanism that can be used in a consistent way across different reusable class libraries." While entirely in favor of robust error handling, I have serious doubts that exceptions will engender software that is any more robust than that achieved by other means. I am concerned that exceptions will lull programmers into a false sense of security, believing that their code is handling errors when in reality the exceptions are actually compounding errors and hindering the software.

To illustrate my concerns concretely I will examine the code that appeared in Reed's article. The code (page 42, October 1993) is a Stack class template. To reduce the size of Reed's code for presentation purposes, I have made two changes. First, instead of throwing Exception objects, my version simply throws literal character strings. The detailed encoding of the exception object is irrelevant for my purposes, because we will see no extraction of information from an exception object. Second, to avoid having to break long lines of source text, I have abbreviated the identifier current_index to top.

Reed's code follows. Spend a few minutes studying it before reading on. Pay particular attention to its exception handling. [Hint: Look for any of the classic problems associated with delete, such as too few delete operations, too many delete operations or access to memory after its delete.]

Original Code for Stack Template

```
template <class T>
class Stack
{
  unsigned nelems;
  int top;
  T* v;

public:
  unsigned count();
  void push(T);
  T pop();

  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
};

template <class T>
Stack<T>::Stack()
{
  top = -1;
  v = new T[nelems=10];
  if( v == 0 )
    throw "out of memory";
}

template <class T>
Stack<T>::Stack(const Stack<T>& s)
{
  v = new T[nelems = s.nelems];
  if( v == 0 )
    throw "out of memory";
```

```
  if( s.top > -1 ){
    for(top = 0; top <= s.top; top++)
            v[top] = s.v[top];

    top--;
  }
}

template <class T>
Stack<T>::~Stack()
{
  delete [ ] v;
}

template <class T>
void Stack<T>::push(T element)
{
  top++;
  if( top == nelems-1 ){
    T* new_buffer = new T[nelems+=10];
    if( new_buffer == 0 )
            throw "out of memory";
    for(int i = 0; i < top; i++)
            new_buffer[i] = v[i];
    delete [ ] v;
    v = new_buffer;
  }
  v[top] = element;
}

template <class T>
T Stack<T>::pop()
{
  if( top < 0 )
    throw "pop on empty stack";
  return v[top--];
}

template <class T>
```

```
unsigned Stack<T>::count()
{
  return top+1;
}

template <class T>
Stack<T>&
Stack<T>::operator=(const Stack<T>& s)
{
  delete [ ] v;
  v = new T[nelems=s.nelems];
  if( v == 0 )
    throw "out of memory";
  if( s.top > -1 ){
    for(top = 0; top <= s.top; top++)
            v[top] = s.v[top];
    top--;
  }
  return *this;
}
```

My examination of the code is in three phases. First, I study the code's behavior along its "normal", exception-free execution paths, those in which no exceptions are thrown. Second, I study the consequences of exceptions thrown explicitly by the member functions of Stack. Third, I study the consequences of exceptions thrown by the T objects that are manipulated by Stack. Of these three phases, it is unquestionably the third that involves the most demanding analysis.

Normal Execution Paths

Consider the following code, which uses the copy constructor to make a copy of an empty stack:

```
        Stack<int> y;
        Stack<int> x = y;
        assert( y.count() == 0 );
        printf( "%u\n", x.count() );
<outputs>
        17736
```

The object x should be made empty, since it is copied from an empty master. However, x is not empty according to x.count(); the value 17736 appears because x.top is not set by the copy constructor when copying an empty object. The test that suppresses the copy loop for an empty object also suppresses the setting of top. The value that top assumes is determined by the contents of its memory as left by the last occupant.

Now consider a similar situation with respect to assignment:

```
Stack<int> a, b;
a.push(0);
a = b;
printf( "%u\n", a.count() );
```

<outputs>

```
1
```

Again, the object a should be empty. Again, it isn't. The boundary condition fault seen in the copy constructor also appears in operator=, so the value of a.top is not set to the value of b.top. There is a second bug in operator=. It does nothing to protect itself against self-assignment, that is, where the left-hand and right-hand sides of the assignment are the same object. Such an assignment would cause operator= to attempt to access deleted memory, with undefined results.

Exceptions Thrown by Stack

There are five explicit throw sites in Stack: four report memory exhaustion from operator new, and one reports stack underflow on a pop operation. (Stack assumes that on memory exhaustion operator new returns a null pointer. However, some implementations of operator new throw an exception instead. I will probably address exceptions thrown by operator new in a later column.)

The throw expressions in the default constructor and copy constructor of Stack are benign, by and large. When either of these constructors throws an exception, no Stack object remains and there is little left to say. (The little that does remain is sufficiently subtle that I will defer it to a later column as well.)

The throw from push is more interesting. Clearly, a Stack object that throws from a push operation has rejected the pushed value. However, when rejecting the operation, in what state should push leave its object? On push failure, this stack class takes its object into an inconsistent state, because the increment of top precedes a check to see that any necessary growth can be accomplished. The Stack object is in an inconsistent state because the value of top indicates the presence of an element for which there is no corresponding entry in the allocated array.

Of course, the Stack class might be documented to indicate that a throw from its push leaves the object in a state in which further member functions (count, push and pop) can no longer be used. However, it

is simpler to correct the code. The push member function could be modified so that if an exception is thrown, the object is left in the state that it occupied before the push was attempted. Exceptions do not provide a rationale for an object to enter an inconsistent state, thus requiring clients to know which member functions may be called.

A similar problem arises in operator=, which disposes of the original array before successfully allocating a new one. If x and y are Stack objects and x=y throws the out-of-memory exception from x.operator=, the state of x is inconsistent. The value returned by a.count() does not reflect the number of elements that can be popped off the stack because the array of stacked elements no longer exists.

Exceptions Thrown by T

The member functions of Stack<T> create and copy arbitrary T objects. If T is a built-in type, such as int or double, then operations that copy T objects do not throw exceptions. However, if T is another class type there is no such guarantee. The default constructor, copy constructor and assignment operator of T may throw exceptions just as the corresponding members of Stack do. Even if our program contains no other classes, client code might instantiate Stack<Stack<int> >. We must therefore analyze the effect of an operation on a T object that throws an exception when called from a member function of Stack.

The behavior of Stack should be "exception neutral" with respect to T. The Stack class must let exceptions propagate correctly through its member functions without causing a failure of Stack. This is much easier said than done.

Consider an exception thrown by the assignment operation in the for loop of the copy constructor:

```
template <class T>
Stack<T>::Stack(const Stack<T>& s)
{
  v = new T[nelems = s.nelems];       // leak
  if( v == 0 )
    throw "out of memory";
  if( s.top > -1 ){
    for(top = 0; top <= s.top; top++)
      v[top] = s.v[top];              // throw
    top--;
  }
}
```

Since the copy constructor does not catch it, the exception propagates to the context in which the Stack object is being created. Because the exception came from a constructor, the creating context assumes that no object has been constructed. The destructor for Stack does not execute. Therefore, no attempt is made to delete the array of T objects allocated by the copy constructor. This array has leaked. The memory can never be recovered. Perhaps some programs can tolerate limited memory leaks. Many others cannot. A long-lived system, one that catches and successfully recovers from this exception, may eventually be throttled by the memory leaked in the copy constructor.

A second memory leak can be found in push. An exception thrown from the assignment of T in the for loop in push propagates out of the function, thereby leaking the newly allocated array, to which only new_buffer points:

```
template <class T>
void Stack<T>::push(T element)
{
  top++;
  if( top == nelems-1 ){
    T* new_buffer = new T[nelems+=10]; // leak
    if( new_buffer == 0 )
      throw "out of memory";
    for(int i = 0; i < top; i++)
      new_buffer[i] = v[i];          // throw
    delete [ ] v;
    v = new_buffer;
  }
  v[top] = element;
}
```

The next operation on T we examine is the copy construction of the T object returned from pop:

¤

```
template <class T>
T Stack<T>::pop()
{
  if( top < 0 )
    throw "pop on empty stack";
  return v[top--];                  // throw
}
```

What happens if the copy construction of this object throws an exception? The pop operation fails because the object at the top of the stack cannot be copied (not because the stack is empty). Clearly,

the caller does not receive a T object. But what should happen to the state of the stack object on which a pop operation fails in this way? A simple policy would be that if an operation on a stack throws an exception, the state of the stack is unchanged. A caller that removes the exception's cause can then repeat the pop operation, perhaps successfully.

However, pop *does* change the state of the stack when the copy construction of its result fails. The post-decrement of top appears in an argument expression to the copy constructor for T. Argument expressions are fully evaluated before their function is called. So top is decremented *before* the copy construction. It is therefore impossible for a caller to recover from this exception and repeat the pop operation to retrieve that element off the stack.

Finally, consider an exception thrown by the default constructor for T during the creation of the dynamic array of T in operator=:

```
template <class T>
Stack<T>&
Stack<T>::operator=(const Stack<T>& s)
{
  delete [ ] v;                    // v undefined
  v = new T[nelems=s.nelems];      // throw
  if( v == 0 )
    throw "out of memory";
  if( s.top > -1 ){
    for(top = 0; top <= s.top; top++)
      v[top] = s.v[top];
    top--;
  }
  return *this;
}
```

The delete expression in operator= deletes the old array for the object on the left-hand side of the assignment. The delete operator leaves the value of v undefined. Most implementations leave v dangling unchanged, still pointing to the old array that has been returned to the heap. Suppose the exception from T::T() is thrown from within this assignment:

```
{
  Stack<Thing> x, y;
  y = x;                           // throw
}                                  // double delete
```

As the exception propagates out of y.operator=, y.v is left pointing to the deallocated array. When the destructor for y executes at the end of the block, y.v still points to the deallocated array. The delete in the Stack destructor therefore has undefined results — it is illegal to delete the array twice.

An Invitation

Regular readers of this column might now expect to see a presentation of my version of Stack. In this case, I have no code to offer, at least at present. Although I can see how to correct many of the faults in Reed's Stack, I am not confident that I can produce a exception-correct version. Quite simply, I don't think that I understand all the exception-related interactions against which Stack must defend itself. Rather, I invite Reed (or anyone else) to publish an exception-correct version of Stack. This task involves more than just addressing the faults I have enumerated here, because I have chosen not to identify all the problems that I found in Stack. This omission is intended to encourage others to think exhaustively about the issues, and perhaps uncover situations that I have missed. If I did offer all of my analysis, while there is no guarantee of its completeness, it might discourage others from looking further. I don't know for sure how many bugs must be corrected in Stack to make it exception-correct.