Standard Exceptions
X3J16/91-0116
WG21/0049

Jerry Schwarz
jss@lucid.com

Sept. 20, 1991

1 Motivation

Ever since X3J16 agreed to incorporate exception handling there has been discussion of whether there should be classes and functions in the standard to support general exception handling. This paper says how a believe this question should be addressed.

I propose that a new header file named `<xmsg>` be added to the standard and that it declare the classes and functions proposed here.

My goals in formulating this proposal (in priority order) are

- Programs that simply want to print an error message and exit when an exception is thrown should be able to do so easily with some chance that the error message will give a meaningful indication of the error.
- User libraries should be able to extend the standard mechanisms so that (a) is also readily accomplished with them.
- A program that wants to determine with more precision the kind of error that occured and recover appropriately should be able to do so.
- The burden imposed on the rest of the library in order to use the mechanism should be minimal.
- Implementation of the mechanism should be easy.

What I have in mind for is that the following should be a plausible way to wrap C++ code that otherwise ignores exceptions.

```
int main(int argc, char** argv)
{
    try {
        real_main(argc,argv);
        return 0 ;
    } catch ( xmsg& m ) {
        cerr << "exiting  because of exception: "
             << m.why << endl ;
        return 1 ;
    }
}
```

If something as simple as the above is to work, then all exceptions thrown by the standard library should be derived from **xmsg** and **xmsg** should contain an insertable value which I will assume is a string. I don't think there is much more that can be meaningfully included in **xmsg**. Some people have proposed an integer error code (analogous to errno) be part of xmsg. But I don't agree. **vm.why** should already contain in human readable form whatever information would be provided by an integer error code. Additional discrimination required by the program should be accomplished by catch clauses on classes derived from **xmsg**.

2 xmsg

The goals of the previous section lead to a class

```
class xmsg {
  public:
    xmsg(string msg) ;
    string why() ;
    void raise() throw(xmsg);
} ;
```

The absence of a default constructor encourages the rule that every **xmsg** should contain a meaningful message.

x.why() is the string used to construct **x**. That is **xmsg(s).why()==s**.

raise adds no functionality but is included as a convenient hook for debugging. It is defined by

```
void xmsg::raise() { throw *this ; }
```

Details of exceptions to be raised by particular classes will generally be contained in the descriptions of those classes. For example, a class `ios::failure` has already been proposed to deal with exceptions in the stream classes. This should be derived from `xmsg`.

3 Allocation

Probably the most common "mistake" made in C++ programs is to assume that evaluation of a `new` expression does not result in a null pointer. Checks for this condition quickly leads to unreadable code and serve no useful purpose. Because there is usually no sensible recovery. The easiest way to "fix" such programs is to change the behavior of `operator new` so that it never returns a null pointer. When the system runs out of space a `new` expression should throw an exception. When I proposed this in Lund and the discussion was generally favorable although there was some hesitation.

Worries concerned situations in which some more space can be made availble to `operator new`. I believe such recovery actions should be done by a function called from `operator new` rather than by the caller. At Lund I thought that the RM specified `set_new_handler` as a way to specify the recovery actions. I have since discovered that `set_new_handler` is a cfront feature not mentioned in the RM.

The current static mechanism for replacing the global `operator new` is unsatisfactory. Its major problem is that since the choice is made statically at linktime, a library cannot reasonably define its own. Because it might conflict with the `operator new` required by another library or the library user. I believe a dynamic mechanism for replacing `operator new` is required. But that is beyond the scope of this document.

All the RM(5.3.3/10) currently says is

> Any form of `operator new`() may indicate failure to allocate storage by returning 0(the null pointer). In this case no initialization is done and the value of the [`new`] expression is 0.

I propose to replace this paragraph by

> Any form of `operator new`() may indicate failure to allocate storage by throwing an exception. If it returns 0 the effect is implementation defined.

But I have never written any code that relies on the current behavior. People who have such code might desire that this paragraph not be changed.

In any event I assume that the behavior of the default **operator new** will be to throw an exception. I propose to have a class

```
class xalloc: public xmsg {
    xalloc(string msg, size_t size) ;
    size_t requested() ;
    void raise() ;
};
```

The base **xmsg** would be initialized by some combination of **msg** and **size**. I don't think the standard should prescribe the form of the message but something along the lines of

```
msg + ": Insufficient space to allocate "
    + itos(size) + " bytes"
```

might be plausible.

Since the exception is going to be thrown when the system runs out of space the question arises as to where to find the space to construct the **xmsg** and (more importantly) any space required to throw an exception. The answer is that the allocator must hold back sufficient space to enable successful construction and throwing the exception. The easiest way to hold back the space for the **xalloc** is to allocate it as a static object. If that is done, the message can't be made to depend on the amound of space requested. Which is why I don't want to prescribe the contents of the message.

In general **operator new** has to be careful that any operations it does don't end up in recursive calls. This may be particularly delicate if the normal exception mechanism mechanism would normally use **operator new** to allocate space. To cope with this I override pvxmsg::raise and require that pvxalloc::raise may be called from a global **operator new** and will throw itself.

Suppose you want to give a more specific error when the program runs out of space while trying to allocate for class Foo. You might write

```
void* Foo::operator new(size_t size)
{
    try {
        return ::operator new(size) ;
    } catch ( xalloc& x) {
        xalloc("in Foo: " + x.msg, x.size).raise()
```

```
    }
}
```

Unfortunately the string concatenation is likely to result in a recursive call to `operator new`. The general technique of catching an exception and adding information to the message is a good one, but it may be advisable to avoid it with `xmsg`.

4 Miscellaneous Conditions

Another class I propose

```
class xassert : public xmsg {
    assert(string msg, string file, int line);
    const string file ;
    const int line ;
}
```

A form of the `assert` macro should be available that throws an `xassert` when its condition was false rather than printing a message and calling `abort`, as the C standard requires. I propose this simply replace the definiton of `assert` contained in the C library.

As the RM stands now there are a large number of undefined actions that can occur at runtime. I have in mind things like dereferencing a NULL pointer. In some environments these are detectable and in such environments a reasonable action would be to throw an `xassert` (or other `xmsg`). I'd like to encourage implementations to do this without mandating it for specific runtime errors. Perhaps we should put something in the library like

```
template <class T> T& xref(T* p)
{
    assert( !p ) ;
    return *p ;
}
```

One undefined situation where I I believe it is reasonable to require implementations to throw an `xmsg` is failling off the bottom of a function that is supposed to return a value. There is no runtime cost (because the program isn't supposed to do this) and the space cost would seem to be minimal.

A related issue is `errno`. I'm not sure what to do about it. I don't want to mandate massive changes in the C library, but on the other hand the current situation is so unpleasant that I don't like leaving it alone either. Any suggestions would be appreciated.

5 Name space

One problem with the proposals contained in this paper are that they constitute a moderate amount of name space pollution. So it might better to do something like like

```
class stdx
{
    class msg { ... } ;
    class alloc : public msg { ... } ;
    class assert : public msg { ... } ;
}
```