Using C++ Exceptions in Commercial Quality MacOS Code

Steve Sisak <sgs@codewell.com>

This paper describes some techniques for using C++ Exceptions in commercial quality MacOS code, including issues related to toolbox callbacks, library boundaries, AppleEvents, and multi-threading.

Introduction

Modern C++ provides many powerful features that aim to make code more reusable and reliable, but in many cases, due to its UNIX roots, these features conflict with equally important features of commercial quality MacOS code, such as toolbox callbacks, multi-threading and asynchronous I/O. C++ Exception Handling is definitely an example of this.

What are exceptions?

Exception handling is a formal mechanism for reporting and handling errors which separates the error handling code from the normal execution path . If you are unfamiliar with how C++ exceptions work, you may want to check out Chapter 14 of *"The C++ Programming Language"* by Bjarne Stroustrup or any of the other excellent texts on the topic.

Why are exceptions necessary?

"Exceptions cannot be ignored" - Scott Meyers

One of the problems in designing reusable code is deciding how to communicate an error that occurs deep within a library function back to someone who can handle it. There are several conventional ways for library code to report an error, including:

- Terminate the program
- Return an error value
- Set a global error flag
- Call an error handling function
- Perform a non-local goto (longjmp)

Let's look at each in detail:

Terminating the program

While unconditionally terminating a program as the result of an error in input may be considered acceptable traditional UNIX programs, it is generally not a good idea in an application meant for use by human beings.

Returning an error value

Returning an error or setting a flag is somewhat better, but these kind of error results are often ignored, either because the programmer was lazy or because a function that returns an error is called by another which has no way to report it. Both of these methods are also limited in the amount of information they can return. Further, return values must be meticulously passed back up the calling stack, and global flags are inherently unsafe in a threaded environment.

Calling an error handler

Calling an error handler function is reliable, but while the function may be able to log the error, it must still resort to one of the other mechanisms to handle or recover from the error.

Unwinding the Stack

So we're left with non-local goto, which is basically how exceptions are implemented—except with help from the compiler. C++ exceptions extend the setjmp / longjmp library functions commonly used by C programs by guaranteeing that local variables in registers are handled properly and destructors for any local objects on the stack are called as the stack unwinds.

Because this technique does not rely on an explicit action by the programmer to propagate errors up the call stack, exceptions are always propagated unless the programmer takes an explicit action to prevent it. Also, because an exception is an object, it is possible for the library developer to return far more information just an error code and to add information without modifying client code.

What's wrong with C++ exceptions?

In a nutshell: lack of standardization.

Like many aspects of C and C++, the implementation of exceptions has been left as an implementation detail to be defined by compiler vendors as they see fit. As a result, it is never safe to throw a C++ exception from a library that might be used by code compiled with a different compiler (or a different version of the same compiler, or even the same version of a compiler with different compile options).

As a result of this:

- Exceptions cannot be thrown out of a library.
- Exceptions cannot be thrown out of a toolbox callback.
- Exceptions cannot be thrown out of a thread.

Each of these cases fail in subtly different ways:

Throwing out of a library

In the first case, there is no guarantee that both compilers use compatible representations for exceptions—the C++ standard does not define a format for exceptions that is supported across multiple compilers.

Exceptions are objects and there is no standard representation for C++ objects that is enforced across compilers. This is also why it's not feasible to export C++ classes from a shared library.

IBM's System Object Model (SOM), used in OpenDoc and Apple's Contextual Menu Manager, solves this problem for objects quite robustly (it is even possible to mix objects and classes implemented in different languages like C++ and SmallTalk), but there are still issues which would require a "System Exception Model" as well.

As a platform vendor, Apple could have saved us a lot of work here by specifying an exception model that all compiler vendors would agree to implement. If fact it appears that they began to implement an Exceptions Manager as part of the PowerPC ABI but didn't finish it—so we're stuck with the current state of incompatibility.

Throwing from a toolbox callback

Many MacOS routines use allow the programmer to specify callback routines which will be called by the toolbox during lengthy operations or to give the programmer more control than could be encoded on routine parameters. Unfortunately, it is not possible to throw an error from a callback and catch it in the code that called the original toolbox routine.

This is because there is no way for the toolbox clean up any resources that may have been allocated by the toolbox before calling the callback. In this case it is necessary to save off the exception data, return an error to the toolbox, and then re-throw the exception when the toolbox routine returns to its caller. But, C++ provides no supported way to save off the exception currently being thrown and RTTI does not provide enough access to extract all data from an object of unknown type, so again, we must roll our own.

A few toolbox managers provide for error callback functions which are not required to return. While it should be possible to throw an exception from one of these callbacks, there are issues that you should be aware of. Specifically, some compilers implement so-called "zero-overhead" exceptions which use elaborate schemes of tables and tracing up the stack to restore program state without needing to explicitly save state at the beginning of a try block. This code often gets confused by having stack frames in the calling sequence that the compiler did not generate causing it to call terminate() on your behalf. (CodeWarrior's exceptions code also does this if you try to step over a throw from Jasik's Debugger -- you can work around this by installing an empty terminate() handler)

Throwing exceptions from a thread

C and C++ have no notion of threading or accommodation for it . For instance, the C++ standard allows you to install a handler to be called if an exception is thrown and would not be caught, but you can only install one such handler per application and it is technically illegal for this routine to return to its caller so there is no easy way to insure that an uncaught C++ exception will terminate only the thread it was thrown from rather than the entire program. (It is possible with globals and custom thread switching routines, but tricky to implement.)

Interactions between threads and the runtime can also rear up and bite developers in even more interesting and subtle ways: for instance, in earlier versions of CodeWarrior's runtime, the exception handler stack was kept in a linked list, the head of which was in a global variable. If exceptions were mixed with threads and the programmer did not add code to explicitly manage this compiler-generated global, the exception stacks of multiple threads would become intermingled, resulting in Real Bad Things™ happening if anyone actually threw an exception.

What we need is a standard way to package an exception so it can be passed across all of these boundaries and handled or re-thrown without losing information.

As any Real Programmer™ knows, good Macintosh programs should be scriptable so that your users can do stuff the programmer didn't think of, and recordable, so that users don't have to have intimate knowledge of AppleScript to record some actions, clean up the result and save it off for future use.

You may also know that if you want to write a scriptable and recordable application and you're starting from scratch, the easiest way to do it is to write a "factored" application — where the application is split into user interface and a server which communicate with AppleEvents.

In a past life, I've written about how using AppleEvents is a convenient way to make your application multi-threaded by using the AppleEvent to pass data from the user interface to a server thread. [MacTech Dec '94] Further, you may know that the AppleEvent manager provides a data structure that can hold an arbitrary collection of data (AERecord).

What you may not know (thanks to the fact that it's relatively hidden in the AppleScript release notes, rather than in Inside Macintosh or a Tech Note) is that AppleScript provides a relatively robust error reporting mechanism in the form of a set of optional parameters in the reply of an AppleEvent which can specify, among other things, the error code, an explanatory string, the (AEOM) object that caused the error, and a bunch of other stuff.

Putting this all together, if we define a C++ exception class that can export itself to an AERecord, we can both return extremely explicit error information to a user of AppleScript (or any OSA language) and provide a standard format for exporting exceptions across a library boundary . And since an AERecord can contain an arbitrary amount of data in any format, the programmer is free to include any information he or she wants in the exception. Anything the recipient doesn't understand will be ignored.

Implementation Details

Following are some excerpts from an exception class and support code which do just this. Full source for a simple program using this code is provided on the conference CD. The exception mechanism is actually implemented as a pair of classes: Exception and LocationInCode and a series of macros which provide a reasonably efficient mechanism for reporting exactly where an error occurred and returning this information in the reply to an AppleEvent.

Using this mechanism, it is not only possible to throw an error across library boundaries, but also between processes or even machines.

Detecting and Throwing Errors

The implementation of the Exception classes is divided between two source files: Exception.cp and LocationInCode.cp. The class Exception is the abstract representation of an exception. It has 2 subclasses: StdException and SilentException.

If you look at these two files, you'll notice that most of the functions that are involved in failure handling are implemented as macros in Exception.h which evaluate to methods of another class, LocationInCode — for instance, FailOSErr() is implemented as:

```
#define FailOSErr
  GetLocationInCode().FailOSErr

#define GetLocationInCode()
  LocationInCode(__LINE__, __FILE__)

class LocationInCode
{
  LocationInCode(long line,
    const char* file) ...

  void Throw(OSStatus err);

  inline void  FailOSErr(OSErr err) const
  {
    if (err != noErr)
    {
      // CW Seems not to be sign extending w/o cast
      Throw((OSStatus) err);
    }
  }
}
```

So that the expression:

```
FailOSErr(MyFunc());
```

Evaluates to:

```
LocationInCode(__LINE__,
  __FILE__).FailOSErr(MyFunc());
```

While this seems needlessly complex, there is a good reason for it involving tradeoffs between speed, code size, and some "features" of the C++ specification.

Specifically, the obvious way to implement FailOSErr() is:

```
#define FailOSErr(err) if (err) Throw(err)
```

The problem here is that the macro FailOSErr() evaluates its argument twice. This means that, in the case of an error, MyFunc() will be called twice — clearly not what we want.

Here is one place that C++ can help us out — we can implement FailOSErr() as an inline function:

```
inline void FailOSErr(err)
{
  if (err != noErr)
  {
    Throw(err, __LINE__, __FILE__);
  }
}
```

Since C++ inline functions are guaranteed to evaluate their arguments exactly once, this solves our problem. Further, it makes it possible to have overloaded versions of FailOSErr which take different arguments, for instance a string to pass to the user, so you can write:

```
FailOSErr(MyFunc(), "Some Error message")
```

The problem is that, once you implement this and try to access the file and line information, you will discover that, thanks to the way __FILE__ and __LINE__ are defined, all errors are reported as occurring in Exception.h — which is clearly less than useful. You would think that the C++ standards committee would have updated the way that these macros work or provided a more robust mechanism for reporting the location of an error in code, but they didn't.

The solution presented here is a compromise. By instantiating the LocationInCode class from a macro, we insure that __FILE__ and __LINE__ evaluate to a useful location in the user's code, rather than the exceptions library. Also, by using a class, we can reduce code size by allowing the methods of TLocationInCode to call each other without losing the actual location of the error.

An added benefit of this approach is that, in the future, we could replace the implementation of LocationInCode with one that used MacsBug symbols or traceback tables in the code instead of relying on the compiler macros.

Also, note that FailOSErr() and the constructor for LocationInCode are declared inline to maximize speed, but then call an out-of-line function (Throw) to minimize code size in the failure case.

At any point in handling an error you can add information to an Exception by calling Exception::PutErrorParamPtr or Exception::PutErrorParamDesc. For instance if you were in an AppleEvent handler and wanted to set the offending object displayed to the user, you could write:

```
try
{
  // whatever
}
catch (Exception& exc)
{
  exc.PutErrorParamDesc(
    kAEOffendingObject, whatever, false);
  throw;
}
```

These routines also take a parameter to tell whether to overwrite data already in the record -- this is useful to insure that the first error that occurred is the on reported to the user.

Insuring Errors are Caught

Because it isn't safe to throw C++ exceptions across a library boundary, we need a mechanism to insure that all errors are trapped and properly reported. Unfortunately, unlike Object Pascal, we can't just call CatchFailures() to set up a handler — the code which might fail must be called from within a try block.

Also, because C++ effectively requires catch blocks to switch off the class of the object thrown and doesn't support the concept of 'finally' like Java, this master exception handler can end up containing quite a lot of duplicated code.

In order to minimize code size, the static method Exception::vStandardizeExceptions() provides a way to have a function called from within a block that will catch all errors and convert them to a subclass of Exception. If you plan to support other exception classes, such as the ones in the C++ standard library, you would modify this function to do the right thing.

```
OSStatus Exception::vStandardizeExceptions(
  VAProc proc, va_list arg)
{
  StdException exc(GetLocationInCode());

  try     // Call the proc
  {
    return (*proc)(arg);
  }
  catch (Exception& err)
    // Exceptions are OK
  {
    throw /*err*/;
  }
  catch (char* msg)
  {
    exc.PutErrorParamPtr( keyErrorString,
      typeChar, msg, strlen(msg));
  }
  catch (long num)
  {
    exc.SetStatus(num);
  }
  catch (...)
  {
  }

  if (LogExceptions())
  {
    exc.Log();
  }

  exc.AboutToThrow();

  throw exc;

  return 0;
}
```

There are several other convenience routines, all of which call through Exception::vStandardizeExceptions(), which capture all exceptions and convert them to an OSErr or write them into an AppleEvent. For instance the following can be used by an AppleEvent handler to catch all errors and return them in the event:

```
OSErr Exception::CatchAEErrors(
  AppleEvent* event, VAProc proc, ...)
{
  va_list arg; va_start(arg, proc);

  OSStatus status;

  try
  {
    status = vStandardizeExceptions(
      proc, arg);
  }
  catch (Exception& exc)
  {
    status = exc.GetOSErr();

    if (event && event->dataHandle != nil)
    {
      if (status != errAEEventNotHandled)
      {
        // AppleScript has an undocumented "feature"
        // where if we put an error parameter in an
        // unhandled event, it reports an error rather
        // than trying the system handlers.
        GetLocationInCode().LogIfErr(
          exc.GetAEParams(*event, false));
      }
    }
  }

  va_end(arg);

  if (status <= SHRT_MAX && status >=
    SHRT_MIN)
  {
    return (OSErr) status;
  }
  else
  {
    return eGeneralErr;
  }
}
```

This pair of functions report all errors to the user. (The Exceptions library allows the programmer to install a callback to report exceptions to the user. Note that here we use vStandardizeExceptions to insure that all exceptions are converted to a sub-class of Exception())

```
static OSStatus report_exception(
  va_list arg)
{
  VA_ARG(Exception*, exc, arg);

  exc->Report();

  return 0;
}
```

```
void Exception::ReportExceptions(
  VAProc proc, ...)
{
  va_list arg; va_start(arg, proc);

  try
  {
    GetLocationInCode().FailOSStatus(
      vStandardizeExceptions(proc, arg));
    va_end(arg);
  }
  catch (Exception& exc)
  {
    va_end(arg);

    try
    {
      StandardizeExceptions(
        report_exception, &exc);
    }
    catch(Exception& exc1)
    {
      exc1.Log();
        // don't throw errors in reporting
    }
  }
}
```

Conclusion

Exception handling is useful and practically required in robust code. C++ exceptions have a number of limitations which you must be aware of when you are developing code using operating system features not supported by the language. However, using the techniques described above, these limitations can be overcome.

Bibliography

[1] Bjarne Strousttrup, "The C++ Programming Language" (Third Edition), Addison-Wesley, 1997, ISBN 2-201-88954-4

[2] Scott Meyers, "Effective C++" (Second Edition), Addison-Wesley, 1997, ISBN 0-201-92488-9

[3] Scott Meyers, "More Effective C++", Addison-Wesley, 1996, ISBN 0-201-63371-X

[4] P.J. Plauger, "The Draft Standard C++ Library", Prentis-Hall, 1995, ISBN 0-13-117003-1

[5] James O. Coplien, "Advanced C++ Programming Styles and Idioms", Addison-Wesley, 1992, ISBN 0-201-54855-0