Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Practical C++ Error Handling in Hybrid Environments

Source Code Accompanies This Article. Download It Now.

- cpperror.txt

How to interface your exception-handling code to software components that practice different error-handling methods.

February 05, 2007
URL:http://www.drdobbs.com/cpp/practical-c-error-handling-in-hybrid-env/197003350

Exception-safe code is notoriously difficult to get right in C++. Still, it is the recommended way to go, at least according to C++ gurus. And for a reason. If you write nontrivial programs or libraries in C++, you should probably study exception handling and use it where appropriate.

In the real world, however, exception handling is not always possible, viable, or used by anyone. So what do you do if you need to integrate or interoperate with software that doesn't use exception handling? In this article, I discuss situations where exception handling is not used and why. I then demonstrate how to interface your exception-handling code to other software components that practice a different error-handling method.

Error-Handling Strategies in C++

There are different ways to report and handle errors in C/C++. (You have to consider C because C++ is—more or less—a superset of C and often C++ programs link against C libraries, libraries that expose C APIs, call C code, and are called from C code via global/static functions or function pointers.)

Exceptions are the official error-handling mechanism, of course. However, C++ was initially designed without exceptions and it shows. For example, IO streams don't throw exceptions by default, but set a "fail" bit when something goes wrong. Compilers, being backward-compatible creatures, usually provide switches to turn exception handling on/off. This means that if you call `new` and run out of memory, you either get a NULL pointer or an `std::bad_alloc` exception.

Status/Return codes are another error-handling technique. This is as simple as it gets and provides the lowest common denominator that is always available, even if you port your code to different languages. Status/Return codes foster visibility in the source code of the path of specific errors but at the price of clutter.

`setjmp/longjmp` is an early C language stack-unwinding exception handler that is widely used to handle input errors detected by deeply nested function calls, such as the recursive descent parsing algorithms of source code language translators. Like C++ exception handling, `setjmp/longjmp` eliminates the need to know about and handle error return codes all the way up the recursive ascent. Unlike the C++ `try/catch` mechanism, which has to destroy automatic objects along the way up, `setjmp/longjmp` adds very little runtime and space overhead to a program.

Likewise, `errno` is yet another C error mechanism. It is just a global object that can contain an error code. It is not thread-safe so its raw usage is discouraged in concurrent systems.

Managed Environments/Proprietary Error-Handling Mechanisms

Because Microsoft is the only vendor that provides deep integration to operating system facilities (SEH) or managed environments (COM and CLR) at the C++ level, all of the following mechanisms are Windows specific. This is an important trend. The Mono runtime brings .NET to the cross-platform world, and while it doesn't yet provide a C++ front-end, there is work towards it. All of these error-handling mechanisms integrate and interoperate with each other and with C++ exceptions to some degree.

Get/Set LastError is a glorified `errno` for Win32. It's better than `errno` in the sense that code in other languages can access it through the Windows API. Cross-language, as opposed to cross-platform, code is a big concern of Microsoft.

Win32 Structured Exception Handling (SEH) is the low-level OS mechanism that calls a certain callback when some thread faults. The Visual C++ compiler wraps it for you and implements the C++ exception model on top of it. You can ignore it most of the time, just know that it's there in the bizarre case you need to dig that deep and find out something.

COM/ATL error handling is a different beast. COM was designed to be language agnostic. As such it couldn't rely on any language's error-handling facilities. So, COM came up with its own error-handling facility. COM has an error object that is accessible via various COM interfaces: `ICreateErrorInfo` for creating error objects, `IErrorInfo` for accessing error information, and `ISupportErrorInfo` to check if a certain COM class actually creates instances of the error object. The punch-line is that I don't really know how ubiquitous the COM error object is. When I used COM/ATL a lot, I used plain HRESULTs, which is equivalent to C status/return codes.

In terms of managed (.NET/CLR) exceptions, the .NET/CLR assimilates, subsumes, and encapsulates all other Windows technologies. It provides its own exception model and specific CLR extensions for C++, but also has a unique mechanism for interfacing with pure C++ code. It reportedly just works most

of the time. I've never used it.

When Exceptions Are Not Used (And Why)

There are two metareasons for not using C++ exception handling:

- You can't. This includes C code (or code that's exposed through a C API), code that runs in some managed environment, and legacy systems built using a compiler that doesn't support exceptions properly or not in a cross-platform way.
- You don't want to (or the powers that be don't want to). This includes performance-critical systems where you don't want to pay the overhead of exceptions. It also includes uncomfortable developers who don't want to mess with exceptions in the first place, or the desire to have a single error-handling mechanism across the system.

C++ is the only language I'm aware of that even lets you turn exception handling on/off. In other languages, you either have exceptions or you don't. You can use other mechanisms if you want, but exceptions are there and runtime failures probably trigger exceptions, so you must be ready to catch them.

If you develop code that you want other people to use, you would be wise to consider exposing it through a C API. The C++ Application Binary Interface (ABI) is not part of the C++ Standard. Every compiler vendor changes it with every major version (and sometimes more often). The result is that you can't statically link a C++ library if it was compiled using a different compiler. It is also not possible on a nonWindows platform to load C++ classes directly from a dynamic library. So, if you are not in the business of shipping source distributions and you want to reach a larger audience than people who happen to have exactly your compiler, stick to C.

With managed environments, you don't even have the choice. If your code must run in a managed environment, you must play by its rules. There isn't much point in throwing an uncaught exception in your COM out of a process module. It would never make it across to the calling code without marshalling.

Legacy systems can be big, yet fragile, creatures. The smallest change might break them, they may rely on undocumented APIs or bugs in a specific compiler implementation. Upgrading the compiler or changing the error-handling strategy is typically prohibitively expansive. Chances are you won't be able to persuade the powers that be to upgrade the compiler and potentially destabilize the system just so you can use cool C++ exceptions in your shiny new module.

Exception handling has a runtime overhead, even when exceptions are not thrown. The compiler needs to manage a list of all the stack objects that were constructed to call their destructors when an exception is thrown. This implies at least a space overhead (if the list is prepared during compilation). If the preparation of this list is done at runtime, then there is also a time overhead. Many performance-critical systems have a shoestring budget for CPU cycles and memory bits, and developers can't afford the cost of exception handling.

In general, C++ developers, who are considered hard-core in comparison to other developer, are uncomfortable with C++ exception handling due to the complexity of implementing it correctly. Exception handling in other languages that support garbage collection (Java, Python, C#) is much simpler and developers have no problem using it. Still, exceptions have a lot going for them. They can considerably simplify error handling in C++ programs. However, they are not simple by themselves. Many developers just know that getting C++ exceptions to work is complicated. Many experienced C++ developers never used exceptions in their code because they only worked on projects that fall under the aforementioned restrictions or they learned C++ before exceptions were introduced to the language. In the initial meeting of a new project when it's time to pick an error-handling strategy, inexperienced developers tend to stick to what they know. Exception handling is not as cool as templates, so developers often don't feel that they "have" to use it, just because they never used it before.

Software is complicated. When system architects face crucial decisions such as picking an error-handling strategy, they often prefer a single mechanism instead of two that have to interact. Since many libraries and existing code bases don't leverage exceptions, this single mechanism is often a simple return code, or one of the other nonexceptional mechanisms.

Exception Handling in Hybrid Environments

Error handling is hard. Mixing multiple mechanisms is harder. The only thing that's even harder is explaining why your system doesn't handle errors properly. Different situations call for different solutions.

Sometimes avoiding exceptions is the right thing to do. Here are a few factors in favor of this approach:

- You maintain an existing system that doesn't use exceptions.
- The piece you work on is performance critical (pending profiling).
- Team members are not familiar with exception handling and are not keen on learning.

If you have already made the decision to use exceptions, you can pick and choose your battles. Not every function must throw exceptions and you probably don't want to convert the error-handling style of a large existing code base. Use exception handling in new code and where it makes sense. For example, if there is a complex piece of existing code that's buggy and tends to mismanage resources when errors occur, it is probably a good candidate for conversion. There is a good chance that the design is not optimal to begin with so while you refactor it, you can also convert it to use exception handling. You would be surprised how small the code can get with automatic objects managing cleanup of resources in destructors.

Catch exceptions on the interoperability border. When you need to interface with another component, you must conform to its API of course (unless you define that API, then the other party has to conform). If this API involves returning error codes or setting `errno` or any other error-handling reporting mechanism, you must not leak unhandled exceptions. It doesn't mean that you can't use exceptions in this case. It just means that you should catch your own exceptions internally in each API method/function and map your exceptions to the appropriate API convention.

The reverse is also true. If your code is heavily exception oriented and you call into some nonexception API, you don't have to deal with yet another error mechanism. You can wrap all the calls through this API in an object that converts API error codes to exceptions.

StreamingException

`StreamingException` (Listing One) is an exception class that exposes a stream interface and allows one-line formatting of complex error messages. The design and implementation are interesting and use C++ constructs such as `std::auto_ptr`, the mutable modifier, and (begrudgingly) exception specifications.

```
#ifndef STREAMING_EXCEPTION
#define STREAMING_EXCEPTION

#include <iostream>
#include <sstream>
#include <memory>
#include <stdexcept>

class StreamingException : public std::runtime_error
{
public:
 StreamingException() :
   std::runtime_error(""),
   ss_(std::auto_ptr<std::stringstream>
       (new std::stringstream()))
 {
 }

 ~StreamingException() throw()
 {
 }

 template <typename T>
 StreamingException & operator << (const T & t)
 {
   (*ss_) << t;
   return *this;
 }

 virtual const char * what() const throw()
 {
   s_ = ss_->str();
   return s_.c_str();
 }

private:
 mutable std::auto_ptr<std::stringstream> ss_;
 mutable std::string s_;
};
#endif
```

Listing One

`StreamingException` is derived from `std::runtime_error`. It's good form to derive your exception classes from `std::standard_error` because it communicates your intention (I hope you throw your exceptions at runtime) and it lets the application catch runtime errors from multiple sources (as long as all the other sources follow this guideline, too). In addition, `std::runtime_error` provides the `what()` method that returns a text message with the error description. By default, `what()` returns the message that was passed in the constructor. `StreamingException` overrides the virtual `what()` and returns instead the contents of its stream (it's not called "StreamingException" for nothing). The `StreamingException` constructor takes no arguments and passes an empty message to its base `std::runtime_error` constructor (remember that this message is not used anyway). The constructor then creates a `stringstream` instance on the heap and puts it in a mutable `std::auto_ptr`. If you studied your smart pointers well, you know that `std::auto_ptr` has the unusual property of ownership transfer. When you assign it to another `auto_ptr`, the assignee gets the ownership of the pointed object and the original object is left with nothing. This peculiar behavior is sometimes dangerous and often gets in the way (you can't put `auto_ptr` into a standard container because you will lose the ownership to the container). In this case it is exactly what the doctor ordered. You will soon see why and why it must be mutable.

The destructor is quite empty, but it can't be dropped. The compiler will indeed generate a default destructor for you, but the default destructor doesn't come with an empty `throw()` exception specification. This is required because `std::runtime_error` defines such a virtual destructor. Exception specifications are an annoying misfeature of C++ that specifies what exceptions a method may throw and are part of the method signature. Thankfully, they are optional so you don't see them a lot in the wild.

The templated `operator<<` is where the pedal hits the metal. This is a template method that accepts any streamable `T`. That translates to standard types plus any other type that implements a conforming `operator<<`. The implementation simply streams the `T` argument to its member `stringstream s_`. The return value is a reference to the `StreamingException` class itself. This allows chaining multiple calls to `operator<<`.

Finally, `StreamingException` overrides the virtual `what()` method to return the contents of its `stringstream`. Note the exception specification again.

Example 1 is straightforward. It tries to compare `5` to `3` and throws a `StreamingException` when the comparison fails. Note the dynamic message was constructed in one line and streamed into the exception while throwing it! (And yes, it works.)

```
int main (int argc, char * const argv[])
{
  try
  {
    if (5 != 3)
      throw StreamingException() << 5 << " is not equal to " << 3;
  }
  catch (const StreamingException & e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

Example 1: Using StreamException.

In the catch clause the exception is caught by a `const` reference. This is a good practice, too. Please refer to your favorite C++ book for the reasoning. The catch clause proceeds to print the exception's message via `what()` to the standard output.

It's time to reveal `StreamingException`'s secrets. The thrown exception is a temporary object. The instance that is caught in the catch clause is actually a copy of the original exception. I had a couple of choices to keep the error message intact during this copy. Most of them involved implementing a copy constructor and some of them involved duplication of the error message. Instead, I opted to use `auto_ptr`, which takes care of two issues: It deletes the dynamically allocated `stringstream` on destruction and it transfers the ownership when `StreamingException` is copied. Okay, so why mutable? Well, the caught exception is a `const` reference because the catching code is not supposed to modify the internal state of the exception. However, `auto_ptr` with its ownership transfer semantics does require a change of state. The mutable modifier was invented exactly for this purpose—being able to modify the internal state of an object while preserving its conceptual `const`ness.

Conclusion

C++ is a multiparadigm programming language supporting procedural, object-oriented, generic, and even metaprogramming. C++ also allows multiple error handling strategies to coexist in the same system. Getting it right is not a simple task. Still, it is possible to create complicated error messages in a safe and concise manner.

Gigi is a senior staff engineer with Numenta specializing in object-oriented programming in C/C++/C#/Python/Java with emphasis on large-scale distributed systems. He can be contacted at the.gigi@gmail.com