Error handling and logging

Introduction

In my last article in Overload I presented code to facilitate error logging. I'd like to stay with this theme and look at some of the issues and concerns involved in logging.

When I speak of logging, I normally mean error logging. The actual "error logs" may be errors, or they may be warnings and information messages. An error log is one end of a spectrum, at the other end is so called "trace" information which I will also comment on.

In addition I'll show how to catch some of NT's more difficult exceptions.

Exceptions and libraries

C++ provides us with the exceptions mechanism for raising, well, exceptions. An exception is not necessarily and error, it is some situation that occurs in a program which is so unusual as to be *an exception.* The majority of these cases are errors, i.e. something unusual has happened which is in error.

Exceptions provide us with a rich environment to encapsulate and describe the condition that has arisen. All this information can be placed in an object, and the object throw, like a message in a bottle, which, through some *angelic determinism*[1] will find a suitable handler which can deal with it.

This mechanism is most powerful when considered in the light of re-usable code libraries used in multiple projects. If we build a library of classes to represent financial instruments, we may use the library in big server boxes doing calculation for an investment house, or in a workstation product used by floor traders, or even a quick calculator for hand held machines. In each case we will want to handle errors differently. Hence, we must abstract the exception handling mechanism in such a way that the library user makes the decision on what to do when a condition arises.

But wait! As developers, who will be called upon to fix the problem, we need to know as much, specific information about how the condition arose to enable us to find and fix the problem – typically, the more information we have the quicker we can fix the problem. This is at odds with out wish to abstract the exception.

This is where the exception object comes into the picture. When the condition arises the code which detects it should package all the information it can into an object. The object is thrown and when caught a *policy* decision can be made which determines what happens next.

Separate cause and effect

The above scenario describes the separation of cause (the exception condition being encountered) and the effect of condition (the functionality failing). This maps conveniently into our developer-user view

---

[1] This expression comes from a lecturer of mine who used it to describe the way a non-deterministic automaton reaches a stop state. I use it here to describe the way a catch will be reached without the thrower being aware of where the catch is in the program.

of the world. As the developer we are concerned with the cause, because we want to fix it. Users, on the other hand, are only concerned with the effect.

For example, if during a client-server TCP session the connection is lost and the server query is aborted, the user is only concerned with effect, their query has failed, they do not care why it has failed[2]. However, as a developer, I'm interested in why was the connection lost, what error number was returned, what was the socket state? And so on.

In this scenario, the throw statement is matched with the *cause*, and the catch statement with the *effect*. Within the catch block the effects of the condition are felt and action taken. This is where we choose to deal with the condition, it is where the logging should be placed.

The code I presented in my previous article could be used as a bed-rock layer upon which libraries can be built. Each library would be able, through this bed-rock to log errors. While this is sometimes what we want, it is better to move the error logging to higher levels where decisions on what action is taken, and what is logged can be taken.

Another advantage of separating cause and effect is that we can manage the quantity of information we present. A user does not want to be bothered with a page of statements and a stack unwind, they want a nice simple error message they can report to described what has failed. To us developers a simple one liner like "TCP socket failed" is next to useless, a several pages of messages and stack unwind are just what we want! (A nice big red arrow pointing at a bug would be helpful too, but I leave this as an exercise to the reader!)

Tracing

Tracing is a form of logging, but it does not deal exclusively with errors. My preferred definition of tracing is: "Tracing allows a developer to see inside a program." Most developers are familiar with the idea of adding debug code to a program so they can track what is happening inside during development, but this code is normally removed for release.

If we look beyond software development to the semi-conductor industry, or less abstractly, kitchen appliances and car, manufactures increasingly leave "debug code" in place, and/or provide interfaces through which diagnostics can be exposed. One of my favourite examples are the inspection facilities built into large civil-engineering projects so that bridges, tunnels and building can be inspected once construction has finished.

This is where tracing comes in. We should actively seek to leave inspection code in place in a program so that, when in live operation we encounter a problem we have some diagnostic tools to hand.

Before I go any further will address the point that many people are already muttering: "debug code comes out and allows run-time speed to increase, if we leave in trace code run-times will be hit and footprint increased" I won't try and deny this, run-times will be hit but most of the time tracing is switched off, we can engineer our application so the only extra overhead is an occasional if statement.

---

[2] If the user can do something to rectify the problem the situation is slightly different but the program design must be such that allows a user to "abort or retry".

When tracing is switched on, performance is hit more but this only happens we have an issue to address.

While the run-time overhead argument may of been noticeable back in the days of 6502 and Z80 processors with 64K of RAM, where every cycle and every byte was valued, is the same true in the world of 600Mhz Pentium III's with 128Mb of RAM? Surely, when we specify the machines for our applications we should include capacity for tracing. Few engineering disciplines today would limit service and maintenance opportunities. We accept that a car's engine is accessible through the bonnet, even for those of us who never open it, yet surely the car would be safer is it was welded shut? After all, the garage mechanic can un-weld it when they need to open it!

Trace statements can also be viewed as comments. Reading code peppered with trace statements is even better than reading comment code as we can see the comments executing, we can see the actual path through the code, and any doubts we have about the actual execution path are ellayed.

And some more things

I have not discussed several points that should be considered when designing your exception handling and logging strategy. I hope to cover at least some of them in a future article:

? Standard C++ exceptions : we now live in a world with a standard C++ library of exception objects. While these may not be suitable for every project I think we should all plan to use them at the outset of our design. Only abandon them if you find good reason. Some of the common library function will through exceptions of these types so you cannot ignore them altogether. Twelve months ago, this may not of been my advice, but things move on.

? Internationalisation : when designing error messages it is worth considering what languages you will need to log errors in. This is particularly important for user facing messages, developer specific information and trace information is probably not worth translating. In reality, I wager, most software, has only one target language[3]. Microsoft tackle this problem with a resource file which resides in a separate DLL. While I praise their attempt I believe the Microsoft solution to event logging and internationalisation of log messages is overly complex and lacks any cross-platform merit.

? Error codes : it is worth assigning error codes to specific messages. Creating an include file which #defined the first error as 1 and continues upwards is the wrong thing to do – this file will inevitable be included in every other file, so whenever you add a new error the whole project will need rebuilding[4]. A better solution is shown by Oracle, Sybase and other database vendors. Here, areas of functionality are assigned short text codes, e.g. SQL for query language functionality, NET for network functionality and so on. Within each area a set of numeric codes is allocated, so we have errors like SQL0001, SQL0002, etc.

---

[3] I state this not as an objective of software but as a reality.

[4] This is the exact example John Lakos gives (*Large Scale C++ Software Design, 1996*) of a practice that will lead to problems on large projects.

? Keep a central record of error messages your system can issue : this is invaluable when documenting the system and running a help desk to support the system. Even where the software does not represent a commercial product it reduces the burden on anyone who has to support the system by living with a pager and/or mobile phone!

? When devising an error code scheme, and an error message document make sure they are easy to work with and expand. The last thing you want is a developer saying: "It's too much hassle to add a new error message and number for this case, this existing code and message is almost right so I'll just reuse that." For once, we don't want to re-use code. Re-using this code will lead to confusion for the technical author and help desk, let alone if the text is translated to another language where it can't do double service!

Now for some code.....

One of the problems with handling exceptions with C++ under Windows NT is that the existence of two exception mechanisms. The C++ mechanism which I'm sure all Overload readers know and love, and the NT *structured exception* mechanism, this is a little like a UNIX signal, when NT encounters an exception (e.g divide by zero, memory corruption) it raises a structured exception.

An NT structured exception is an unsigned integer which identifies the condition. It may be caught with a C++ catch-all (catch(...)) but so is everything else. Further, the stack management semantics are different, an NT structured exception will not unwind the stack. This makes it difficult to compile code with both mechanisms in use – indeed Visual C++ with issue a warning message at the very least.

The solution, as shown here, is to install an exception translator. This is done by registering a function callback using the _set_se_translator function:

```
_set_se_translator( Translator )
```

Once installed, NT will call the translator whenever it hits a structured exception. (Shown in the listing NtSeTranslator.h & .cpp.) This provides us with an opportunity to create an object to represent the exception and throw this. A problem occurs as NT calls the function for each stack frame as it clears the stack, hence the translator will be called multiple times so we cannot embed information within the exception object as a new one is created and thrown at each stack level.

We can however, catch a translated exception using regular C++ catch semantics and obtain access the error code. We can also stop the stack unwind when the catch occurs.

The code is in two parts. Firstly the actual translator function, which is an large ugly switch statement. Secondly, the translator class – SeTranslator. SeTranslator implements an allocation is initialisation metaphore to install the translator when an object of SeTranslator is installed. Normally, I declare one of these just inside the first try block of my code.

```
try
{
     // install a translator
     Accu::SeTranslator translator;
     int *ptr = NULL;
```

```
        std::cout   << "Goodbye crewl world..."
                    << *ptr << std::endl;
        return 0;
    }
    catch(Accu::StructuredException& exp)
    {
        std::cerr   << "Something happened: "
                    << exp.what() << std::endl;
        return 1;
    }
```

I suspect that UNIX signals could be wrapped in such a way to similarly throw an exception when received. This raises an interesting cross platform handling technique.
I also include an exception class, StructuredException (in listing StructuredException.h and .cpp), derived from std::exception which is thrown when an NT structured exception is throw.
Finally, over the time I have been using these classes the most useful facility is one which is not immediately obvious. Within the debugger, place a break point inside the translator function. In the event of a structured exception (e.g. de-referencing a null pointer) the translator is called and the breakpoint encountered resulting in a debugger trap at the point after the error occurred with a full stack trace available and watch windows.

Conclusion

I hope I have convinced you that when thinking of exceptions we need to think in terms of cause and effect, the cause maps to the developers view of the exception and the effect maps to the users view of what happened.
Under Windows NT the effect can sometimes be all to obvious and the cause unknown, however, by bringing the NT exception handling mechanism into the C++ world we can, once again, deal with cause and effect.