Error logging and patterns

Introduction

Back in Overload 32 Francis suggested that his lack of experience with larger systems made him ill-equipped to review design patterns. The implication, of cause is that design patterns are for large systems. He then went one to throw down the gauntlet of a challenge to anyone, to explain some patterns. I didn't set out to pick up the gauntlet – in my head this article was sketched – but hopefully I can show Francis where patterns are applicable on a small scale.

This article sets out to look at error logging and present a solution which is scalable from small to large systems. In the process of tackling the problem several well known patterns are used.

Error handling is universal to small and large systems, and whatever mechanism you use to handle errors there is always a need to log errors. In my experience, error handling and logging is one of the key reasons why small solutions often don't scale upwards. In a 500 line program you can send error reports to standard error, or a message box where you please. In a 50,000 line program which is distributed across multiple machines and compiled from re-usable libraries something more substantial is required.

The solution I present here is designed to scale from small systems to large systems.

All the patterns referenced here are from Design Patterns by Gamma et al. Often referred to as *Gang of Four patterns* because the book was written by four people (who, of cause, are the *Gang of Four* – or GoF if you like TLAs). While not the first work on patterns it is the work which introduced the software community at large to them.

The problem

There principal requirements are:

1. The error log should be easily accessible and easy to use : developers should be encouraged to log early, log often.
2. It should be possible to log to multiple destinations (sinks.)
3. It should be possible to add and remove sinks at run time.

In a console application we traditionally send error messages to the console on stderr/cerr, and in a GUI we typically send the errors to message boxes for the user to read and dismiss. In larger systems we may log errors to a file, database, remote error logging component or an OS based log, e.g. the NT Event log and UNIX syslog. We may want to log errors to different places at different times, although it is hard to beat a simple text file for audit purposes. In more exotic configurations it may be desirable to dynamically change your logging sinks: e.g. when connected to the internet send error reports to the home site.

I've recently been working on lights out servers which, ideally, run without any user intervention. In these cases the last thing we want is a message box appearing saying "Abnormal program termination – press OK to continue"; we'rd much rather the program fail, log a message to the log and allow the watchdog to restart it.

You will need.....

To build this system you will need:

? One chain of responsibility design pattern[1] : each destination sink is listed in a map, the message is passed to each sink which has the opportunity to do something with it.

? Two singleton design patterns[2] : all messages for logging are channelled to a single point, from here they are distributed to the various sinks.

? One very small strategy pattern[3] : this is used to reduce the amount of code needed when iterating over the map.

? One proxy pattern : using a proxy output stream allows messages to be streamed to the log.

? One namespace : suitable for implementing a sub-system.

? One critical section[4] : this allows the model to be used in multi-threaded systems.

? Classification of messages system[5] : here divided into Error messages, Warning messages (potential errors), Information messages (not an error, but something worth noting) and Debug messages (something the developer thought might be useful to know.)

? Pointers to members : not a typo but a lesser know aspect of C++, documented in Stroustrup[6] section 15.5

? Initialisation of non-local objects : see Stroustrup, section 9.4.1

The code presented here has been written with Microsoft Visual C++ 5.0 on Win32. I see no reason why it should not compile first time on version 6.0. I've compiled it under egcs 2.91.66 but haven;t yet linked and tested it. Other dialects of C++ should, namespaces provided, require little or no changes.

The implementation also contains one *anti-pattern* but you may wish to avoid this particular example.

Error logging is a sub-system

Error handling should be build at all levels of a system from the bottom up. Hence, any logging mechanism should inhabit one of the lowest layers. This layer exhibits several characteristics:

? High integrity code : the code will not be executed often (we hope) but when it executes we expect it to work perfectly, the last thing we want is an error in the error logging. Paradoxically, the most important code is some of the least used. As usual the solution is to keep the code short and simple.

? Good performance and limited risks : if the system is failing we may not have much time left, further we don't know what has gone wrong, it would be better to avoid allocating memory (we may be out of memory), performing GUI actions (the GUI may be stalled), and so on. Of cause each system will be different. When we have an error to report we want to do it fast and at minimum risk.

? The log should be accessible from any point in the program or libraries : an error may occur at any point, if the logging mechanism is wrapped in class we must have a pointer or reference to the object before we can log. Either we make this a global or we pass it as a parameter.

? The sub-system should be stable : so we can build it into our libraries.

? The system should allow for expansion : if corporate policy mandates that all logs should be made to a central server we should be able to add this with minimum source code changes.

Overview of patterns

Singleton

This is probably the easiest pattern to understand and one of the most widely used. The pattern name is an accurate summary of the pattern: the is only one of this thing. Usually, the thing is a class, so a singleton pattern typically occurs where there is only one instance of a particular class.
In this example, there may be many log sinks, but there is only one point of logging. Behind this is a singleton object, the SinkStore which implements the functionality, it would be pointless to have more than one SinkStore object. Because SinkStore is hidden behind a namespace I have not enforced the singleton pattern with program code.
The simplest way to ensure you have a singleton in your system is to add a static member to the class. Initialise it to zero and have the class constructor check it is zero before incrementing it. If the member is ever non-zero in the constructor this cannot be the only object of this class so throw and excepetion or assert out.

Proxy

A second singleton exists in this system. The LogStream class which acts to as a proxy, it provides access to the log functions via the stream out operator. Often a proxy is used to restrict access to a function or object, but in this instance we are using a proxy to provide syntactic sugar, making it easier to log messages.
A proxy is best summed up as: “a stand-in for the real image.” Distributed technologies (DCOM and CORBA) frequently use proxies on a local machine to hide the location of the real target. A proxy may do very little, as in this case, or it may perform some work (e.g. marshalling parameters for network transfer in DCOM), but it never performs the actual task you requested.

Chain of responsibility

In this pattern a chain of handlers – here implemented with a map but any traversable structure may be used – is formed. A message, or other object, is passed to each handler in turn, allowing each to do something with it. Each handler implements functionality without the object being aware of what is being done. Hence we decouple the object being acted upon from the action performed, adding actions becomes easy.
In this implementation each message is passed to each link in the chain. One variation – which is used in Microsoft ATL for windows messages – is to have each handle return a “handled flag.” When a handler returns true the chain is terminated.
(It is interesting to contrast chain of responsibility with visitor. Broadly speaking, in a chain the data visits the algorithms, in a visitor pattern the algorithms visit the data.)

Strategy

In a strategy we know what we want to do, we can define an interface to represent what we want to do, but we don’t want to get our hands dirty with the actual details. We are concerned with the bigger

picture. In this case we are concerned with ensuring that the message is logged, we don't care about how or where.

A strategy allows us to vary the algorithm we are using while keeping the big picture the same. The standard C++ library containers are good examples of this. We know how to iterate over them, and iterating over each one is the same, but how each container does it is different from container to container, and even implementation to implementation.

Strategy relies on encapsulating the interface to a process in such a way that the process can differ without effecting the general technique.

Design

The sub-system presented here replies on a singleton logger. This is implemented as set of functions in a namespace. As a namespace we can provide global access to the logging functions without the dangers present in global variables.

The logger namespace presents an interface which hides an implementation[7] using a map. When a message is sent for logging it is passed to every handler in the map allowing it to be handled any log sink.

All elements in the map must be derived from the abstract base class LogSink. This provides the interface which allows the chain and strategy pattern to work.

```
class LogSink
{
public:
	explicit LogSink() {}
	virtual ~LogSink() = 0 {}
	virtual void Debug(const std::string&) const = 0;
	virtual void Information(const std::string&) const = 0;
	virtual void Warning(const std::string&) const = 0;
	virtual void Error(const std::string&) const = 0;
};
```

Elements of the map are created on the heap by the main program and adopted by the sub-system. Once *adopted* the main program can choose to forget about them, they will be deleted when the application is closed. If however, we decide to remove one of the elements we may request it to be *orphaned*, at which time is removed from the map and returned to the caller for deletion. To make this possible all elements are named.

Messages may be sent to the logger either by direct function calls: Debug, Information, Warning and Error; or by way of a stream class which may accept elements of any type which operate with the standard streams. (You may add additional overloads to provide syntactic sugar for other string types you may be using.) A single output channel is provided – accessed through MsgOut() – for sending these classes to the logger.

```
namespace Log {
```

```
        void AdoptSink(const std::string& name, LogSink* const);
        LogSink* OrphanSink(const std::string& name);
        void Debug(const std::string&);
        void Information(const std::string& msg);
        void Warning(const std::string& msg);
        void Error(const std::string& msg);
}
```

Multi-threaded systems are allowed for by guarding critical sections. This code should present a minimal burden on single-threaded systems.

Implementation

MsgOut stream

The Log::MsgOut() function returns a reference to the logger stream (LogStream struct) to which messages may be sent. This stream could have been exposed as a global variable – within the namespace – in a similar way to which std::cout and std::cerr are exposed. However because the stream is implemented as a class there is the possibility that there may be an attempt to write to the stream before it is initialised. Hence it, must be initialised before it is used and this causes a problem. Imagine a global object hitting and error in it's constructor and attempting to log before main() is called.

To quote Stroustrup: "In principal, a variable defined [at global scope] is initialised before main().... in their declaration order." If we could be sure that out object was declared first we would be home free, but we can't[8]. However, if we make the object a static variable in a function it is guaranteed to be initialised before first use. Hence the MsgOut() method.

```
// log.h
struct LogStream { };
LogStream& MsgOut();


// log.cpp
LogStream& MsgOut()
{
        static LogStream out;
        return out;
}
```

Four message classes exist, one for each message type which are derived from std::strstream. Each is used in conjunction with a operator<< function to log messages to the MsgOut stream. The classes themselves inherit all their interfaces and implementation from std::strstream.

```
struct DebugMsg : public std::strstream { };
struct InformationMsg : public std::strstream { };
```

```
struct WarningMsg : public std::strstream     { };
struct ErrorMsg : public std::strstream { };


LogStream& operator<<(LogStream&, DebugMsg&);
LogStream& operator<<(LogStream&, InformationMsg&);
LogStream& operator<<(LogStream&, WarningMsg&);
LogStream& operator<<(LogStream&, ErrorMsg&);
```

Since LogStream and the message classes are empty a good compiler should be able to optimise them away altogether.

Strategy for chain handlers

When a message is received it is passed to each handler in turn. Each element is free to implement an empty function if it wishes, or take whatever action it considers most appropriate.

The actual iteration is performed by the WriteMsg function regardless of the actual method being called; the function implements the iteration, the actual mechanics, are irrelevant, or to put it another way, the strategy is to call a method on each handler. This is implemented using a pointer to a member.

As it's name suggests, a pointer to a member is not a function call itself, but is a pointer, to a function call. Unlike pointers to functions it is not an absolute memory address but an offset into the v-table of an object, it also ensures the object called is available. It is perfect for this application where all the objects are derived from a single base class, and all the functions which may be called are similar virtual (indeed pure-virtual) functions of the base class. Importantly the member functions are interchangeable.

What is actually present, is a very small strategy pattern: the members form "a family of algorithms" which are "interchangeable" which "lets the algorithm vary independently from... use." In future a "new operation" can be defined "without changing the classes on which is performed." (Quotes are taken from GoF's summary of strategy).

Handler elements

Handlers must be derived from LogSink class to ensure a common interface. They may be placed in the Log namespace if you wish but this is not essential. They are added to the chain using the AdoptSink function and removed with the OrphanSink function. When a handler is removed the responsibility for deleting it passes to the callee.

I provide three example handlers:

- standard error handler : writes to stderr/cerr
- file error handler : writes to a file you name
- box error handler : which puts a Windows message box on screen, while being Win32 specific this shows how different message types can be handled differently.

Here I must plead guilty to an anti-pattern: Cut-and-paste programming[9]. As you will observe, the four functions implemented by each class are very similar, and the three examples look very similar. I used cut-and-paste programming.
Also, the examples use inline methods for brevity. In general I do not condone inline methods (but that's another article!) so I suggest you separate them into interface and implementation files before use.

SinkStore

SinkStore class encapsulates the list of handler. By implementing this as a class and using the static local trick we are sure it is initialised before first use, and that the destructor is called at close to free any resources. LogSink objects owned by the store will also get a chance to free resources (e.g. file handles, data base handles) when their destructors are called.
SinkStore only exists in the implementation of the log sub-system, i.e. the Log.cpp file. Therefore we are free to change the implementation without incuring any side effects, or even the need to recompile dependent code. Given that the log sub-system is in the bottom layer and permeates all aspects of the system this should reduce the need to rebuild all the source code after a tweak.

Problems

? The operator<< functions must get the message string from the std::strstream classes. However, we cannot depend on the string being returned from str() to be null terminated. Nor can we use operator>> to put contents into a std::string because the delimiter will split the message.
We could copy the contents from the stream into a buffer, either a static one, or an expanding std::string. However, I balked at the cost of this intensive operation. Copying to a static buffer would also run the "how big is the biggest buffer" problem, while a dynamic buffer could be allocated to a large enough size I don't want to dynamically allocate memory because, as I've pointed out, we may be low on memory.

My less than perfect solution is to put an end of string character (null) into the stream and take the string out. This has the side effect that the stream returned to the caller is different to that passed in. The recompense here is that it should stop someone using the same stream twice, but, this is only apparent at run time.

? Using the default map ordering in the sink store results in the handler being ordered alphabetically. This is not a particularly great idea, ideally we want the safest handlers to get the message first and more risk ones to get it later; for example "Database" would be called before "StandardError" yet, the chances of a database update failing is far greater than standard error.
Although I don't advocate naming your handlers in alphabetical order of safety a simple fix is to prefix your safest handler with "!", e.g. "!StandardError" would handle an message before "Database". The more robust solution is to rank the handlers in order of safety and have the map ordered by safety level.

? There is no default handler, so if an error occurs before any are installed it will not be logged anywhere. This is deliberate on my part. A default handler may easily be installed in the constructor of SinkStore but each project should think about where the default sink sends it's messages.

Example program

The example program, main.cpp simply demonstrates how to install handlers, remove them and send messages to them. The first message sent to the logger is lost because no handler is installed. Because the handlers are created on the heap, and the logger's scope is effectively global there is no danger of handlers being left in the chain when they pass out of scope.
The code fragment below demonstrates how it all fits together:

```
Log::InformationMsg msg;
Log::Debug("Creating std error sink");
Log::AdoptSink("Console", new Log::StdError());

time_t t = time(NULL);
tm *tt = gmtime(&t);
msg << "Standard error has been adopted at "
        << tt->tm_hour << ":" << tt->tm_min << std::endl;

Log::Debug("Creating file error sink");
Log::AdoptSink("File", new Log::FileError("Log.out"));
msg << "File error has been adopted" << std::endl;
Log::Debug("Removing std error sink");
Log::LogSink* s = Log::OrphanSink("Console");
delete s;
```

And then there was Microsoft.....

Underlying almost all the code I write are Microsoft libraries. Most of the time this is fine, but at times I hit a problem and despair at the lack of design patterns and broader consideration these libraries can display.
Occasionally the libraries encounter a run-time error. And occasionally when an error is encountered they cannot return an error code. And occasionally they don't throw a C++ exception, nor do they create an NT structured exception. Instead they start off Microsoft's own error handling chain. But this chain is not the flexible one documented in GoF or here but a hard coded one.
Lets take an example, suppose, somewhere, somehow, a pure-virtual function is called on the abstract base class. Obviously this should not happen. But it does. In this event the libraries call _purecall which in turn calls _amsg_exit, which writes a message to the console (for console apps) or, conveniently, puts up a message box with an OK button on GUI apps.

The problem is two fold: first, the message box with an OK appearing halts the program until the user clicks it, if the program must die it would be preferable to die silently and have it automatically restarted; second, the real solution is to fix the bug but if there is no way to capture the error....
Microsoft seem to recognise the problem and provide _CrtDbgReport which does allow you to install a handler, but only in the debug build version – and you still can't remove the existing ones. Needless to say, in the real-life version of this problem the error occurs about once a month in the release build and less often in a the debug build!
In short, the Microsoft error logging is not suitable for 24x7 production environments. Needless to say, if Microsoft adopted the design and patterns outlined here the problem would not exist. I could simply remove the GUI handler and replace it with my own alternative.

Mixing patterns

Patterns are often used in combination, e.g. abstract factories to produce singletons; visitors implementing strategies; proxies and just about anything. Where one pattern starts and ends can sometimes be difficult to see. In truth, it doesn't matter where one starts and ends. Your program will still compile if you cross two patterns. (John Vlissides *C++ Report* column is frequently filled with crossed patterns.)
A pattern presents a way of looking at a problem. At least one of the patterns in this example was discovered after the code was written and article drafted. You may think of a pattern as presenting a *view* onto the *model* which is the source code.
Patterns are often talked about as a language. A pattern language is not a set of rules and syntax like C++ or Java but a set of terms, the same way we may talk about medical language or even bad language – it represents some subset which has particular meaning. Like any language they are open to interpretation and regional dialects. What is important is that they provide a language framework for talking about problems at a higher level.

Conclusion

The code here is written in an extendable fashion so that new handlers can be added at any time while the core implementation can also be changed without side effects[10].
Using a namespace subsystem instead of a class removes the need to pass a pointer or reference to all points in the code where an message may be logged. Hence compile time dependencies are reduced and parameter lists are free from clutter. (Those who attended Bjarne Stroustrup's talk at the ACCU April meeting will recognise this immediately of something which is not an object.)
This article set out to demonstrate how design patterns can help software scale from the smallest to largest systems by looking at a real life problem. The pattern implementations are somewhat smaller than those in GoF but embody the same principals. I think most of us encounter and use design patterns far more regularly than we realise, even where these patterns are documented the tendency is for the author to present the refined form of the pattern, in real world code patterns are often present in less refined forms and are harder to see, indeed the original developer may not of realised they where using a pattern. This, I think, is why Francis came to believe they where for large systems.

[1] Gamma, Helm, Johnson & Vlissides, *Design Patters*.

[2] Gamma, 1995

[3] Gamma 1995

[4] I've used the one I presented in Overload 31, which is interface compatible with the version in Overload 33

[5] This classification is a sub-set of those available using UNIX syslog functionality; adding the excluded classifications is left as an exercise to the read!

[6] Stroustrup, The C++ Programming Languages, third edition, 1997

[7] Separation of interface and implementation is one of the building blocks of modern software architecture and patterns. I am sure most Overload readers will recognise the mantra immediately, but is it a pattern? I am sure it is but has it ever been documented as such?

[8] Even if we could guarantee this was the first object initialised what would happen when another sub-system came along with another object that demands to be the first initialised?

[9] Brown, Malveau, McCormick & Mowbray, Anti-Patters, 1998

[10] John Vlissides in the *C++ Report* (February 1998) said : " A hallmark – if not the hallmark – of good object-oriented design is that you can modify and extend a system by adding code rather than hacking it." I don't think any other single sentence in all the software literature has had as much effect on my coding and designing as this sentence. The design shown here embodies this philosophy.