contents

credits & contacts

Overload Editor:
Alan Griffiths
overload@accu.org
alan@octopull.demon.co.uk

Contributing Editor:
Mark Radford
mark@twonine.co.uk

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: The Buzzword Adoption Pattern?

In my last editorial (in Overload 60, called "An Industry That Refuses to Learn") I asserted that the software development industry has not made significant progress in the last quarter of a century. This assertion provoked enough of a response to fill the letters page in the following issue. I'm pleased about that, but at the same time, not so pleased. I'm pleased because I managed to provoke people into putting pen to paper – or rather, in this day and age, putting fingers to keyboard. I'm not so pleased because the response was one of overwhelming agreement, which is unfortunate because it suggests that any hopes I may have had that my experience is the odd one out, are false.

Once again it's my turn to write an editorial, and in search of inspiration, I dug out Overload 62 and reread Alan Griffiths' editorial "The Value of What You Know". In that editorial, Alan recounts how a colleague asked him how to return a `NULL` string – because in C the colleague would have represented the string using `const char*` and therefore could, and would, have returned `NULL`. The developer just expected to get a simple answer because it never occurred to him that in the context he was working, a different solution may have been appropriate; in other words, returning `NULL` may or may not have afforded the best set of tradeoffs in the given situation.

Anyway, why am I going on about this? Well, it's because I have observed on many occasions over the last few years, that when a developer comes up with a solution to a problem, they think the problem is solved and get on with implementing whatever it is they've come up with. Like the developer in Alan's story, they don't stop to consider that implementing a particular solution has its own set of consequences – or, putting it another way, they don't consider that there are tradeoffs to be considered.

A recurring example of this is speeding up the lookup process in a data structure in memory, by keeping an index in memory in addition to the data. This approach makes the simple trade of using more memory in return for a gain in speed. Whether or not the tradeoffs are acceptable depends very much on the execution environment. For example, if the structure holds enough data to take up (say) thirty percent of a computer's memory, then the index is likely to be sufficiently large to have an impact on both speed and memory requirements. It should be noted that if measures must be taken to speed up element lookup, then there is an implication that the structure is likely to be large. Further, even if the structure will be large, an index will not be of any benefit if most of the elements searched for are near the starting point for the lookup (typically the beginning of the structure). The upshot of all this is that the solution using indexing is only a good idea if:

1. There is enough memory to support it
2. The overhead of referencing the index will not impose too much overhead on lookup speed, too much of the time

What we're heading towards here is Pattern territory. It's worth extending the discussion to consider Patterns, because the original idea of Patterns was that any particular Pattern captures not only a problem and a solution, but also the tradeoffs that must be accepted if the solution is adopted. The problem of speeding up lookup in a data structure and solving it using indexing, may or may not qualify as a pattern (there are other factors that are beyond the scope of this discussion). However, there is an analogy to be drawn, because indexing is a solution to the problem, but only if a certain set of tradeoffs – i.e. the two cited above, at least – are acceptable.

The concept of a Pattern originates in "The Timeless Way of Building" by the Architect Christopher Alexander, first published in the late 1970s. The idea was imported into the software development community in the early 1990s. However, it was the publication of "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the "Gang of Four" (or "GoF"), that brought Patterns to the attention of the software development community at large, focusing on twenty-three patterns from the domain of object-oriented design. Over the years this book has become known as the "GoF book".

Unfortunately the GoF book, in its attempt to make Patterns more accessible, also (partly due to its presentation of Patterns, and partly due to the way it has been read by the development community at large) accidentally popularised some misconceptions. For example:

- Patterns are for object-oriented design, and there are twenty-three only, no more and no less
- A Pattern is a configuration of classes that works in more than one place.
- Patterns are invented

Whereas, in reality (respectively):

- Patterns occur at all stages of the development process
- A Pattern captures a problem, a solution, and the tradeoffs involved
- Very important: Patterns are harvested from existing practice/experience

In my Overload 60 editorial "An Industry That Refuses To Learn", I expressed my concerns that by bringing Patterns to the attention of the development community at large, the GoF book has lead to them being hijacked and turned into a buzzword – not because of any fault of the GoF, but because it is in the nature of the industry to do this. I am finding more and more, that having the phrase "Design Patterns" on my CV is becoming advantageous when applying for contract work, when it comes to playing the inescapable "buzzword bingo" with the agencies. I have a recollection from an interview, of being asked if I was familiar with "The Patterns Book"; I recall telling the interviewer, yes, I've read "Software Patterns" by James Coplien. Naturally that approach didn't get me anywhere. In passing, and for the record, I regard "Software Patterns"[1] to be a much better candidate for being *The* Patterns Book than the GoF book, because it is much more likely to leave the reader with an understanding of what Patterns are and what they can offer.

Above, I cited indexing as an example of speeding up lookup in a data structure subject to certain tradeoffs being acceptable. The industry's hijacking of Patterns has meant that the tradeoffs element has been lost. Given that the most effective means of learning is by example, the loss of tradeoffs means that a valuable example of considering tradeoffs when considering solutions has been lost.

Anyway, at this point I'd like to move on to a different but related topic, and this time on a more positive note. In this editorial and in my previous one mentioned above, I have talked about the how the software development industry has hijacked concepts and practices and turned them into buzzwords. Well, it seems this has – by accident rather than design – lead to an area of improvement, and I can write about something other than doom and gloom.

Back in the early days of the desktop PC, programmers typically just sat down at the computer and wrote code, testing their work by some sequence of random actions that involved running the program they were working on and seeing what happened. Their practices were unfortunately a far cry from the rigor of their mainframe counterparts. Today the computers used as both desktop workstations and as servers, are much more advanced that the PCs of ten and twenty years ago. Further, the software that runs on them is much larger in scale and much more complex. However, all too often, the software development practices have not advanced. Until now, that is...

It seems that one of the age-old practices of the mainframe developers of old, namely that of unit testing, has resurfaced! You've probably heard mention of "Test-Driven Development" (aka "TDD" or "Test-First Development") in various places over the last few months. Well, a friend of mine went for an interview recently, and was asked what he knew about this practice. He got the job, and it played a part that he'd read Kent Beck's book "Test-Driven Development"[2], which went down well with the interviewer. I have noticed recently myself, this phrase is beginning to be mentioned occasionally when the agencies reel off their list of buzzwords.

Granted, TDD is a bit more than just unit testing (it advocates that a piece of code's tests should be written before the code itself is written), but that's not really the point. The point is, there must be some ironic twist in the fact that unit testing is finding its way back into mainstream workstation/server programming, simply because someone thought up a buzzword – or rather a buzz-phrase – to associate with it. Don't get me wrong though, I'm quite happy that TDD is becoming trendy. Sadly it was for the wrong reasons that TDD came to the attention of this development community at large, but it can only have a positive effect on the quality of software.

In "The Value of What You Know", Alan finishes with the sentiment to the readers: "I'll have to trust you'll have your own story to tell". Well, here I've told (one of) mine – or rather, I've started in one place and followed a line of thought to where it led me (there is an element of thinking out loud in this editorial). I've looked at Patterns – or rather, all that has gone wrong for Patterns – being adopted by the development community. I then moved on to look at how by a happy accident TDD has been adopted, but in this case to the benefit of the projects on which it is used.

Perhaps we should learn from experience with TDD and take stock of practices that we would like to see adopted more widely, and then sharpen our skills in coming up with buzzwords and/or buzz-phrases that are sufficiently catchy for the majority of developers and/or managers. Then, as what happened to TDD happens to other useful practices, maybe the "Buzzword Adoption Pattern" will start to emerge.

Mark Radford
mark@twonine.co.uk

1 Available as a free download from: http://www.bell-labs.com/user/cope/Patterns/WhitePaper

2 Note that Kent Beck is completely open about the origins of unit testing. In his book "Extreme Programming Explained", he asserts that XP is built on the premise that its practices are of proven effectiveness.

Copy Deadlines

All articles intended for publication in *Overload 64* should be submitted to the editor by November 1st 2004, and for *Overload 65* by January 1st 2005.

The *Encapsulate Context* Pattern

by Allan Kelly

Prologue

Encapsulate Context was born after a query to `accu-general` in Summer 2002. I don't think I was looking to write a pattern but I'm glad I did – in retrospect I wish I had begun writing patterns earlier. However, writing patterns is hard work and time consuming. Between the first draft and the version you are about to read 18 months elapsed. The paper may bear my name but it wouldn't be the paper it is without the help of many others.

The version here hasn't changed in several months and is the same as that printed in the EuroPLoP 2003 conference proceedings. This doesn't mean it won't change again, to some degree patterns are living entities that change as people use them and gain experience. So, I wouldn't be surprised if the pattern appears again, elsewhere, with further revisions.

Abstract

A system contains data, which must be generally available to divergent parts of the system, but we wish to avoid using long parameter lists to functions or global data; therefore, we place the necessary data in a Context container and pass this object from function to function.

Audience

Encapsulate Context is principally written for software developers designing and writing programs. The pattern was originally written for C++ developers, however examples have been reported from other languages such as Java and Smalltalk. It is believed that users of any language will find the pattern useful, although C++ developers may find the pattern of particular interest.

By exploring the pattern in depth this paper offers a rigorous explanation of where the pattern occurs, the forces and the consequences of using the pattern. For reference purposes a summary section has been included at the end of the paper. Experienced developers may prefer to read the summary first before reading the entire paper.

Example

In traditional structured programming, global data is minimised by use of function call parameters. This tradition has continued, with some modifications in object-oriented programming. For example:

```
void ProcessMarketTrade(MarketMessage& msg,
                        MarketDataStore& store) {
  if(msg.Trade() == Sell)
    store.Sell(msg.Commodity(),
               msg.Price(), msg.Quantity());
  else
    store.Buy(msg.Commodity(),
              msg.Price(), msg.Quantity());
} // ProcessMarketTrade
```

We now decide that any trade which results in a negative quantity should result in an error message, hence the function `Sell` must have access to the log manager, consequently a handle must be passed down. The code becomes:

```
void ProcessMarketTrade(MarketMessage& msg,
                        MarketDataStore& store,
                        LogManager* log) {
  if(msg.Trade() == Sell)
    store.Sell(msg.Commodity(), msg.Price(),
               msg.Quantity(), log);
  ... as before ...
```

Such changes have a habit of reoccurring, so, when we add a transaction history the code changes again:

```
void ProcessMarketTrade(MarketMessage& msg,
                MarketDataStore& store,
                LogManager* log,
                TransactionHistory& history) {
  if(msg.Trade() == Sell)
    store.Sell(msg.Commodity(), msg.Price(),
               msg.Quantity(), log, history);
  ... and so on ...
```

Several problems are clearly apparent. First the parameter list is growing with a negative effect on comprehensibility, even though the additional code is trivial it increases the bulk. Secondly, we are breaking encapsulation. Initially `Sell` was an encapsulated function, by adding more and more parameters its inner workings are being exposed.

More ominously, we have a ripple effect running through interface and implementation code. The function that calls `ProcessMarketTrade` must itself have access to `LogManager` and `TransactionHistory`, and in turn, the function that calls that function, and so on. Even though these functions will only act as pass-throughs for the handles they are affected.

Less obvious is the capacity for redundant code to enter the system. If at some future date we dispense with the transaction history then removal impacts at least three different functions. To be sure, the temptation would be to disable the code while leaving it in place, hence we simply make it an anonymous parameter in `Sell`:

```
void MarketStore::Sell(Commodity& c,
                       Price& p,
                       Quantity& q,
                       LogManager* log,
                       TransactionHistory&) {
  ....
```

In choosing not to delete the history in full we are storing up complications for future refactorings, we are also half-way to implementing the *Poltergeist* anti-pattern (Brown, 1998).

These problems are exacerbated when a dependency inversion design is adopted. We may decide to recast our market message processing as a *Command* pattern (Gamma, 1994):

```
class MarketMessageCommand {
public:
  virtual void Action(MarketDataStore&,
                      LogManager*) = 0;
  ....
};
```

```
class Buy : public MarketMessageCommand {
public:
  virtual void Action(MarketDataStore&,
                      LogManager*);
  ....
};

class Sell : public MarketMessageCommand {
public:
  virtual void Action(MarketDataStore&,
                      LogManager*);
  ....
};
```

To ensure substitutability each `MarketMessageCommand` must implement `Action` with the same signature as the abstract base class. Consequently commands such as `Buy` are complicated with parameters which are unused. Worse, the potential for ripple effects is magnified across all objects in the hierarchy. If the exchange introduces a programmatic way of signalling transition point in the trading day with an enumeration such as:

```
enum TradingDay {
  Closed, PreOpen, Open, Settlement, Suspended
};
```

A new market message is needed to handle this, but so too is a state variable:

```
class TradingDayChange : public
MarketMessageCommand {
public:
  virtual void Action(MarketDataStore&,
                      LogManager*,
                      TradingDay& activity);
  ....
};
```

Since our new message can change the state activity a new parameter is needed, to maintain a common signature this parameter must be added to `MarketMessageCommand` and all derived classes. Again, we are increasing the length of the parameter list, introducing a ripple effect and adding complexity. Our main loop may look like:

```
int main() {
  MarketDataStore marketData;
  LogManager *log(LogFactory());
  TradingDay exchangeStatus(Closed):
  MessageSource source;
  while(true) {
    auto_ptr<MarketMessageCommand>
                      w(source.NextMessage());
    w->Action(marketData, log, exchangeStatus);
  }
  delete log;
  return 0;
}
```

Faced with the problem of adding yet more parameters we may be tempted to consider global variables. After all, an exchange is open or closed, there is only one instance of such a flag surely? A tempting solution, the exchange status is a simple variable, initialisation is not a significant problem, and being stack based a memory leak is a non-issue.

However, for `LogManager` a global variable is decidedly less tempting. The example above strictly controls the use of log through scope and parameter passing, were the same variable global it could potentially be accessed before creation, e.g. the `MarketDataStore` constructor may choose to log a message.

We would then be forced into the position of trying to enforce creation before use. This is known to be problematic and the best known solution (access through a function) suffers from known issues in multi-threaded systems. Further, the same problems occur in reverse when cleanly ending the program.

While we may be able to survive one or two such global variables we quickly find the number increasing, first the exchange status, then the log manager, what of our transaction history? Have we loaded any DLL plug-ins? Better have a global list of their handles. As we add more global variables it becomes harder to reason about the initialisation sequence for each – particularly important when one makes use of another. It is also more difficult to reason about the internal state of the program because it is dispersed with no central point of reference.

Even with the best will in the world the old issues of globals still exist. Judicious use of namespaces, and careful coding may afford us the luxury of a few globals but the old issues have not gone away, merely repositioned or hidden for a while.

The solutions so far suggested do nothing to improve either the testability of our system or the transfer of components to follow-on projects. Suppose we wish to use our `MarketMessageCommand` in a market simulator. Long parameter lists, and global variables force us to implement plumbing around the hierarchy so we can use the commands.

Likewise, if we wish to write a test harness for our hierarchy, or force test data through the system we must implement the necessary plumbing to support the classes.

Each additional parameter or global variables makes the classes and methods more specific and less of a commodity. Without such specifics, the `MarketMessageCommand` hierarchy implements generic, run-time polymorphic handling of messages. Longer parameter lists increase coupling, tying classes closer to the environment, shorter interfaces are more loosely coupled and result in a more general the class.

The nub of the problem is the ever-expanding parameter list. At first this appears simply unsightly, however, as we can see, the need pass more and more parameters is a real issue.

Problem

Access to common data is important to many systems. Many systems contain data which must be generally available to divergent parts of the system, e.g. configuration data, run-time handles and in-memory application data.

However, we wish to avoid using global data – such data is normally regarded as poor engineering practice. Traditionally the problem is addressed by passing such data as function call parameters but over time parameter list become longer. Long parameter lists themselves have an adverse effect on maintainability and on object substitutability.

While access to such data is a common requirement neither of the two common techniques are without problems. Access to the data is not as trivial as it first appears, and as any system grows the drawbacks of each solution become greater.

Forces

There are several forces that any solution to this problem must accommodate for it to be widely applicable.

1. Substitutability

Software designs based on common interfaces, with object substitutability – either run-time polymorphic or compile-time polymorphic – are restricted in the parameters that can be easily passed to an object because all objects must conform to a common interface with common function signatures to ensure commonality of access – i.e. the Liskov Substitution Principle – LSP (Liskov, 1988, Martin, 1996).

However, where all data is supplied to objects and function via call parameters, if any object requires additional data it must be passed via a call parameter, to keep LSP all similar objects must also accept this parameter even if they have no functional requirement for it.

For an object, changing any function-method call signature, whether by addition, revision or removal breaks LSP. The object in question can no longer be substituted for other similar objects. The compiler should refuse to compile the resulting program. Typically we must either change every class in the same hierarchy to match the new signature, change every call to the function-method, or both.

Having broken LSP we are forced to restore LSP by changing other parts of the system. This creates ripple effects through the code base. A good solution to the overall problem would ensure that LSP is not broken, and consequently, ripple effects within the code base are minimised.

2. Encapsulation

Good software practice values encapsulation, however, traditional solutions threaten encapsulation:

- Over-long parameter lists to function calls reduce encapsulation because the parameters suggest the internal workings to developers.
- Global variables break encapsulation by definition. They are considered poor programming practice, leading to side-effects and increased coupling.
- Within C++ systems there are additional problems associated with instantiation and destruction – particularly in multi-threaded developments. Although C++ namespaces allow better management of globals they do not resolve instantiation and coupling problems.

A good solution would preserve encapsulation thereby minimising side effects and coupling.

3. Coupling to the Environment

The parameters passed to a function, or method, define the state of the system external to the object in question. An object receiving a method call knows its own state (even if this is stateless), what it does not know is the state of the rest of the system, i.e. the context in which it is called. If global data is used it becomes harder to reason about the state of the system at the point of call.

Likewise, a simple function maintains little or no state between calls, the external state is everything, the result of the function call depends on the context in which it is called.

The more tightly coupled an object is to its environment the more difficult it is to use the object in a different setting. Opportunities for using the object in a different environment, e.g. within a test harness, or re-used in a different system, are much reduced. At the same time, the amount of consideration developers must pay the object's environment is increased. Thereby, reducing readability, understandability and maintainability.

A solution that minimised coupling would do much to improve understandibility, maintainability and improve the opportunities for alternative uses.

4. Avoid Data Copying

One solution to the global v. parameter conflict would be to retain a copy of such data in individual objects. Unfortunately, this is not always practical, especially when the system has a large number of small objects and/or objects exist in difference execution threads.

Reasons for not copying pieces of data may include, but are not limited to:

- Data may be changing rapidly, e.g. equity market prices, and needs to be available in several different locations in the program
- Data and operations on the data may overwhelm the class, e.g. a simple command class used in a *Command* pattern may only have one significant method, to additionally store data, handles, and accessors would rob the class of its simplicity.
- Overhead of a copy operation both in terms of time and memory used – this is particularly so if the data is seldom accessed, e.g. command line options.
- Data may be singleton in nature, or encounter problems when copied, e.g. a handle to a log file may be easily copied but we do not wish to store multiple copies of the handle to prevent dangling pointers (or references) when the file is closed. However, use of the *Singleton* (Gamma, 1995) pattern may not be appropriate.

Since these potential solutions are unavailable they represent forces in their own right. Further, as modern systems frequently end up with a large number of small objects these problems are increasing.

Solution

Provide a Context container that collects data together and encapsulates common data used throughout the system.

For example:

```
class Context {
  LogManager* log_;
  ComandLineOptions cmdOpts_;
  ApplicationData* store_;
  ....
};
```

Rather than supply multiple parameters, we supply a Context object. The object acts as a container for program state data, a central repository for widely used data within the system. The Context object provides few, if any, functions itself. The object is passed, or more likely a reference is passed, to functions when they are called – utilising the "parameterize from above" paradigm.

There are typically three types of data found in a context class:

- **Configuration data**, e.g. command line options.
- **Application data**, e.g. market data.
- **Transient run-time data**, e.g. handle to log manager.

The example given here uses one context class for simplicity. While the simplicity of a single context has a lot to recommend it, without careful attention the class may become a kitchen-sink, overwhelmed with any, and all, data in a system. When this happens we start to see the emergence of a *Blob* anti-pattern (Brown, 1998).

To counter the drift towards *Blob* we can split the class into two or more discrete classes, e.g. one for system data and handles with a second for application data (see Figure 1).

Specifically, we can distinguish three types of split:

- **Temporal:** data is separated on the basis of its lifespan, data which is short lived is kept separate from data which exists for

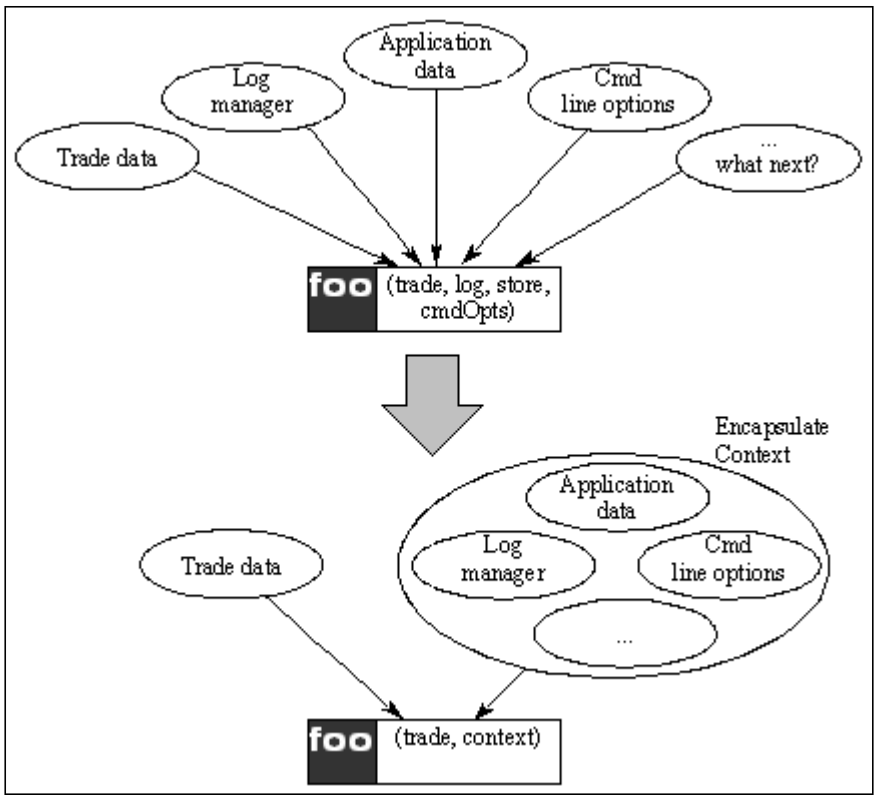


Figure 1: Solution places context data in a single container

long periods. . It is better not to mix transient data with persistent data lest expired data remains in the container.

- **Horizontal:** separating reference data from value data, usually needed when one application becomes large itself, inflating the size of the context.
- **Vertical:** separating the context class into a small hierarchy, usually needed when the same context is needed in a family of programs. This allows for specialisation through inheritance to provide each family member with a specialised Context object and common code to be shared across the family.

Such splits will mitigate the *Blob* tendencies but also detract from the pattern simplicity. Splitting the Context class should also help improve compile times, since we can assume that although some functions will need to be passed all the fragments of the original context, many will require fewer fragments thus reducing dependencies.

However, while it may be desirable to split the Context class for a variety of reasons this can be taken too far. The use of many fine-grained Context objects may return us over long parameter lists.

Thus, any implementation of *Encapsulate Context* pattern should consider the following issues:

- **Is a single Context class the best answer?** The initially simplicity of a single Context may lead to difficulties as anti-patterns emerge.
- **What is the life expectancy of the data?** Bundling short-lived or rapidly changing data together with constant data may lead to confusion or inaccuracies.
- **Is there a family of programs under development?** Is there benefit from creating vertical hierarchy of Context facilitating technology transfers between programs?
- **Are we creating problems by mixing reference and value data in the same context?** Could this data be split horizontally between several Context objects?
- **Are we in danger of creating too many, fine-grained, Context classes?**

These issues must be addressed together as the answers to each question influences the others.

Resolution

Applying this solution to the example given at the start of this paper we get:

```
// MarketContext.hpp
class LogManager;
class CommandLineOptions;
class MarketDataStore;
class MarketContext {
  LogManager* log_;
  CommandLineOptions opts_;
  MarketDataStore* marketData_;
public:
  MarketContext(LogManager*,
                CommandLineOptions&,
                MarketDataStore*);
  LogManager* Log();
  MarketDataStore* MarketData();
  CommandLineOptions& CmdOptions()
                                  const;
};
```

With this context class the presence or absence, of a `TransactionLog` is abstracted to a detail about `MarketContext`.

The class should take a minimal role in the lifetime of enclosed classes, it is better to present these as ready constructed to the class. This removes life-cycle issues from the domain of the context class, and, because enclosed classes are often just references or pointers, the .hpp interface file should only need forward declarations thereby reducing potential ripple effect.

(The decision on whether to use pointers or references to object is outside the scope of this paper.)

Continuing this example the body of the program is refactored:

```
class MarketMessageCommand {
public:
  virtual void Action(MarketContext&) = 0;
  ....
};
int main() {
  LogManager* log(LogFactory());
  CmdLineOptions options(argc, argv);
  MarketDataStore marketData;
  MarketContext context(log, options,
                        &marketData);
  MessageSource source;
  while(true) {
    auto_ptr<MarketMessageCommand>
                    w(source.NextMessage());
    w->Action(context);
  }
  return 0;
}
```

The context provides access to data which otherwise may be made *Singleton*, global or both, for example the `LogManager`.

Variations

- **Provide parent's this pointer**
  The passing of this pointers to worker objects can be seen as a variation on this theme, in effect the calling object is itself acting as a context object for the worker objects. (One consequence of using Context classes is that the need to pass this is usually reduced.)
- **Provide forwarding functions to encapsulated data**
  Rather than expose an entire member class the `MarketContext` class could implement forwarding methods, for example, the `CmdOptions` member could be replaced with:

```
class MarketContext {
  ...
  bool IsVerbose() const {
    return opts_.IsVerbose();
  };
  ... and other forwarding functions ...
};
```

  However, it is best to keep the class as lightweight as possible, to this end, the class exposes the key objects encapsulated rather than implement pass through calls onto the underlying data. It is the underlying class that decides what to expose rather than the context class. Further, although such forwarding functions may be convenient they contribute the tendency for the context class to become a *Blob* (Brown, 1998) so are best avoided.

Consequences

As a result of the pattern, several of the forces detailed above are resolved or balanced:

1. Substitutability

Parameters passed to a function call can be restricted to Context objects containing system state data and parameters which specifically refer to the function call task in hand, e.g. market trades. Function signatures are free of the clutter which can make them fragile – there is no longer a need for every class method in the hierarchy to accept every parameter ever needed.

2. Encapsulation

The Context object effectively compacts the parameter list on a function call signature, thereby abstracting state variables and promoting encapsulation of the function. In addition there is a reduction in ripple effect as function signatures become more stable.

Having relieved the problems of passing parameter to a function the attractions of global data are reduced. Indeed, the Context object provides a natural home for data with characteristics of global variables.

3. Coupling to the Environment

The Context class is encapsulated through its own, well-known, common, interface. This allows the solution to be applied to compile-time and run-time polymorphic designs, using either template metaprogramming or v-table dispatch techniques.

By providing several context classes data is encapsulated along temporal, horizontal or vertical lines further reducing coupling. It is difficult to eliminate all coupling because some classes will always need other classes, to be sure, choosing the granularity of the coupling is a design issue.

Additionally, by separating the classes implementing algorithms, from the plumbing which supplies the data the classes themselves are less coupled and more like commodities, making transfer to other developments easier.

4. Avoid Data Copying

Since the Context class contains common data with little overhead there is no need to copy the data in local objects.

There may be multiple references to the Context object in the system, particularly if multiple threads are being used. Hence some care must be taken to avoid dangling references to Context objects.

In addition there are other beneficial consequences:

5. Reasoning

State data that needs to be shared or retained is factored, objects are left with either transient data or completely stateless. By centralising the core data within a system we have made it easier to reason about the system. We can halt the program and look in one place to see what state the program is in rather than having to look in multiple places.

6. Instantiation

Instantiation issues are simplified because objects must be created before being placed in the context and are subsequently only accessed through the context. Destruction issues are similarly handled because all access is via the context. The life-span of the context can be clearly defined at a high level.

7. Uncluttered Code

Pass-through code and long parameter lists have been minimised, and the potential for future redundant code has been reduced – it is easier to add and remove elements from the Context class. (This may entail a recompile of the whole system when the interface to the Context class is changed but recompilation should be well-defined procedure.)

8. Synchronisation Point

The Context class can provide a useful place to add mutexes for multi-threaded systems. In multi-threaded environments the Context object can hold all shared data, acting as a gatekeeper with mutex control. This is reminiscent of the *Monitor Object* pattern (Schmidt, 2000) with the same potential for bottlenecks if lock access is not carefully considered.

Bottlenecks may be avoided if the data is either immutable (e.g. command line options which do not change), or data elements manager their own locking (e.g. a log manager which implements its own synchronisation) and application data is absent.

However, there are several less desirable consequences:

9. *Blob* Tendencies

As already mentioned, care must be taken as systems develop that a context class does not become a *Blob*. Already in the example given we see the mixing of value data and reference data. Without vigilance context classes may grow to encompass far more data and functionality than is strictly necessary.

Invariably, the context class ends up touching most aspects of the system. It is therefore best-placed low down in the dependency hierarchy of classes – although this can lead to its own dependency inversion problems and small changes necessitate a major recompile of the system.

Once this happens we are in danger of implementing the *Blob* anti-pattern.

Fortunately, change to the Context class tends to be additive in nature so seldom breaks other parts of the system, still, the friction of change is increased. One way to minimise this is to ensure that no operations are placed inside the context class. A second technique is to use multiple Context classes as described above,

however, introducing too many Context classes will introduce some of the original problems we sought to resolve.

10. Hidden Globals

Blind use of Context classes can give rise to an abuse known as "Hide Forbidden Globals" (Green, 2001). This is characterised by a kitchen-sink approach to the Context class where every second variable is listed. Typically we see Context members which are referenced in only a few points within the system, usually such data would be better embedded in specific classes rather than placed in Context.

11. Dominant Sibling

Program families may share a common root Context class, which they embellish through inheritance. In this model the context underpins the common code of the family. If one family member becomes dominant there will be pressure to enhance the common root to facilitate the dominant member. This has a negative effect on the other family members which start to see the common root as a Blob, forcing upon them additional dependencies and complications they do not need.

In the program family we find elements of functional overlap, e.g. a market trading system and a market simulation system. Both may use the `MarketMessageCommmand` and hence rely on the `MarketContext` class as above. As one program, say the simulation, becomes more important and bigger objects start to appear in the command hierarchy which are specific to the one application, eventually, one of these will require some data which is not available in the context class. For immediate simplicity we are tempted to add this into the context. Unfortunately, the trading system now has this data even though it is never used. If continued, over time, the trading system will be inhibited by a Context class which is obscured with unused functions.

More confusing too are the results if the trading system now develops its own specialist message commands, and makes demands for specific fields on the context class.

This is normally an indication that the Context class should be split vertically. We may choose to create a hierarchy of three classes: a common base class, a derived class with simulator enhancements and second derived class with the trading system enhancements.

At this point we may compile different versions accepting either a `SimulatorContext` or a `TradingContext`, or we may choose to down-cast the provided context – assuming that the simulator message classes will only ever be passed a `SimulatorContext` by way of a `MarketContext` handle.

Known Uses

Chutney Technologies Apptimizer (C++)

Apptimizer uses a single Context object to store handles to important system objects, e.g., `Configuration`, `CachedData`, `ConnectionServer`, etc. These system objects are accessed by polymorphic command objects, which receive the Context as a parameter to their `execute()` method.

Reuters Liffe Connect Data Router (C++)

This system uses two context objects, split horizontally. The first holds system data, log manager handles, a configuration cache, COM parameters, and the second holds application data exclusively.

Jiffy XML Database Server (C++)

The Jiffy server has three context objects split along temporal lines. One Context object exists for the length of the program run, this encapsulates process wide context, items such as: log manager handle, command line options and the database store index. A second Context class is used to represent data associated with connections. Each TCP connection is assigned a session context to hold items such as the user id for the connection. Finally, the underlying database from Sleepycat uses its own database-context object to maintain state between database calls.

In this case, the database-context objects are short lived, each one is limited to function call scope (although it will be passed to several underlying functions in turn). A session context lives for the duration of the TCP connection, while the process context is created shortly after the application starts running and is destroyed at the end of the program run.

Enterprise Java Beans

Enterprise Java makes use of Session Beans and Context Beans that encapsulate program state information. Although the objective of Java Beans is to implement component based transaction programming the most of the underlying forces are the same, namely: substitutability of different *beans*, encapsulation of context from server to client and clearly defined coupling.

However, the fourth force, *avoid data copying*, is absent. In the distributed environment for which Java Beans is designed data copying is essential.

Related Patterns

***Command, Chain of Responsibility* and *Objects for States*.**

Although the *Command* pattern is cited here the same principles apply to any design based on the dependency inversion principle using class hierarchies, e.g. *Chain of Responsibility* (Gamma, 1994), *Objects for States* (Henney, 2002), etc. For each of these the hierarchy provides the algorithm while the Context object(s) provide the data.

Singleton

Encapsulate Context may be a useful alternative to *Singleton* (Gamma, 1994) in many program designs.

Observer

Encapsulate Context may be contrasted with *Observer* (Gamma, 1994). Like the Subject in *Observer* the Context class is a central repository of data. Like *Observer* there is a many to one relationship. However, the critical difference lies in the updating mechanics.

The subject in *Observer* knows its observers, when it is updated it will update all its observers. This satisfies the motivation for the pattern that seeks to keep two, or more, objects consistent. Thus, when one *Observer* changes, and hence changes the Subject the other Observers must be informed. In effect, Subject is an active participant in the execution of the program.

In *Encapsulate Context* there is no requirement on the Context class to inform its clients that something has changed. Indeed, it doesn't know who its clients are so it cannot inform them. *Encapsulate Context* keeps the various objects consistent by centralising the data. It is essentially passive during execution.

While there is obvious transformation for turning a Context object into a Subject, and hence *Encapsulate Context* into an *Observer* pattern, and vice versa, there are fundamentally different motivations and forces underlying the two patterns.

Monitor

As noted above (Consequences section), in multi-threaded systems mutex control can be added to *Encapsulate Context* to assist with synchronisation issues. In this the pattern is acting like Schmidt's *Monitor Object* (2000). While this can provide a simple way to synchronise access to resources it is not without a cost.

Firstly, by using the context class as a monitor introduces pressure to perform more processing within the monitor class. This contributes to the *Blob* tendencies already described.

Secondly, the consequences encountered by *Monitor Object* are introduced into the design. Specifically, the liabilities associated with *Monitor Object* need to be recognised, i.e. limited scalability, complicated extensibility semantics, inheritance anomaly and nested monitor lockout.

Readers are strongly advised to read Schmidt before using this *Encapsulate Context* as a synchronisation point.

Arguments Object

This pattern shares much in common with Nobel's *Arguments Object* pattern (Nobel, 1997). The key difference is that Nobel suggests the pattern as a code level pattern for reducing the number of parameters passed to a function, while *Encapsulate Context* advocates using the same paradigm as a high level feature to wrap the state of the system.

Introduce Parameter Object

Both the *Encapsulate Context* and *Arguments Object* patterns resemble Fowler's *Introduce Parameter Object* refactoring pattern (Fowler, 2000). However, Fowler introduces this as only a refactoring pattern without discussion of the issues involved in grouping data or alternative solutions. It is possible to view Fowler's pattern as an application of either *Encapsulate Context* or *Arguments Object* when refactoring code.

Open Arguments

Some of the motivations of *Encapsulate Context* are shared with *Open Arguments* (Patow, 2003). Both aim to provide a consistent interface through which, diverse parts of a system may access parameters. The focus of *Open Arguments* is internal mechanisms of the context object and how this object may support a dynamic set of parameters at run-time. In contrast, *Encapsulate Context* focuses on parameter passing at compile-time. *Open Arguments* considers a parameter block which stored the various parameters, this has clear parallels with the context class in *Encapsulate Context*. The two patterns do not exclude one another, and under the right circumstances may be complementary.

Discussion

Separating Data

At first glance *Encapsulate Context* may seem counter to the principles of object-orientation, this is not so. Instead we are separating the data into that which (a) truly belongs to a given object (e.g. market price and quantity) and (b) that which is owned the system as a whole. There is a casual similarity with the separation of algorithm and container used by the Standard Template Library.

Instantiation Issues

While this paper notes the instantiation problems associated with global objects it does not provide an in-depth discussion or offer detailed solutions. To do so is beyond the scope of this pattern. However, it is suggested that some of these problems can be alleviated by application of the *Encapsulate Context* pattern.

Testing

With designs based on *Encapsulate Context* we may arrange for artificially configured context objects to used for testing. For example, a test harness could create a context object and populate with data to simulate a scenario we wished to test, the test can then be run without to see how the system behaves in these conditions.

Extending this ideas we can imagine two versions of the `MarketContext` class, one of which validates all inputs and one that is optimised for speed. Alternatively, a Context class could load test data to create a specific test scenario, or dump their "state" to file at the end of a test – or in the event of program failure.

Aspect Oriented Programming

Aspect oriented programming may provide an alternative means to resolving some of the forces which produce this pattern. The data within the Context class certainly seems to cross-cut the systems concerned. The logger functionality is both a core example for both Aspect documentation and this pattern. Since C++ does not currently support Aspects, nor are they a standard part of Java, they cannot be regarded as a common solution to this problem and forces.

The main difference appears to centre on the method of passing the context object to the function. This pattern assumes that the context object is passed by way of a function parameter, however, beyond this assumption the concept of bundling the context into one object is still applicable. The key difference is the mechanism for accessing the context object.

Pre and Post Conditions

By their nature, context objects represent the state of the system. This makes them very good places to make uses of pre and post conditions to validate system state. Indeed, developers using context objects should be encouraged to use pre and post conditions.

Use of such pre and post conditions is regarded by many as good programming practice. Used as comments these can help developers reason about the state of the system, used as compiler enforced checks (e.g. macros in C++, conditionals in C#) the system can perform a degree of self validation as well as helping programmers reason.

Pre and post conditions could be placed within the context objects "getter and setter" functions to validate the state of the object, or used by functions accepting context objects to ensure the program is in a suitable state for the function.

Use of such conditions to check state of the system is common practice formal methods systems, e.g. VDM (Jones, 1986) and Z (Wordsworth, 1992). Such languages specify a "state" for the system before and after and operation – the program state in VDM parlance. Further research is need on whether *Encapsulate Context* pattern can be useful in development of formal methods based systems.

Value Data or Reference Data

The solution section, above, notes that care should be taken where reference and value type data is mixed within a single Context object. Such mixing may be a signal that refactoring may be required, and that the Context object should be split horizontally.

However, Context objects observed in actual system frequently mix these data types. While this may indicate poor design it also reflects the fact that Context objects may be required to group various types of data with different reference characteristics. This fact may also indicate that the pattern has been introduced to a system as the result of refactoring and that other parts of the system have not been refactored yet.

Genesis of a Pattern Language – Further Research

Many of the issues raised in the discussion section suggest further variations of this pattern beyond those outlined already. It is also possible to see how, taken together, *Arguments Object*, *Introduce Parameter Object*, *Singleton*, *Open Arguments* and *Encapsulate Context* may represent part of an entire pattern language. We may tentatively label this pattern language *Context Objects*.

For example, *Singleton* could be redefined as an example of *Encapsulate Context* where there is only one instance of the *Context Object*, and the object is accessed via a global variable instead of via parameter passing.

There are four groupings within which to consider variation within the *Context Objects* pattern language:

Access Mechanism

Function parameter passing is used in *Encapsulate Context* to make the Context object accessible. In contrast, *Singleton* uses a global access point. Thread local storage has been suggested as an alternative access mechanism for multi-threaded systems. A further access mechanism, where available, is the Point Cut provided by AspectJ and other aspect oriented languages.

Context Lifetime

While *Singleton*s are generally instantiated for the lifetime of a program run, Nobel's *Arguments Objects* are more ephemeral, being created and destroyed in a short space of time. By extending the consideration of the temporal aspects – described above for *Encapsulate Context* – more pattern variations are possible.

Cardinality of Context

Related to the discussion of lifetime is the issue of cardinality of Context objects. Obviously in cases such as *Argument Object* it is of little importance whether one or one hundred Context objects co-exist. However, in some cases it may be important to limit the number of Context objects within a system, for example, we may wish to limit each thread to one instance of an object, or limit a whole program to one Context object corresponding to the mouse state.

Internal Implementation

Encapsulate Context assumes a fixed internal state where data elements are hard coded and fixed at compile time. In contrast *Open Arguments* allows the content of the Context to change at run time. As already noted both patterns share other similarities and thus may belong to a common language. In this case, the internal representation of data can have a significant effect on system design.

The creation of a *Context Objects* pattern language is beyond the scope of this paper. However, it is clear that such a language could unify existing patterns and probably help identify more patterns.

The author looks forward to hearing about such a project and is more than willing to participate in such an endeavour.

More Examples

The examples presented are given in C++ although it is expected that the pattern is generally applicable to all languages. The author looks forward to hearing of implementations in Java and C# especially.

Summary

In any non-trivial system there will be a number of data elements that are widely used throughout the program, e.g. log manager and the application data model. Typically these will be classes in their own right and accessed through handles (references or pointers.) Since global data is regarded as poor practice it is likely that these handles will be passed by way of function call parameters. However, this technique can soon lead to long parameter lists which are not only difficult to understand but tend to make the program more fragile.

Therefore, we create a context class that encapsulates these data elements, and pass a handle to this object to the diverse functions.

While similar techniques have been suggested by others (e.g. Nobel, 1997, Fowler, 2000) this pattern discusses the forces and consequences when applied system wide. This can bring considerable benefits to a design, but if used recklessly can result in a number of known anti-patterns.

Rather than use a single context class it may be appropriate to design a system with several. These are divided along temporal, horizontal or vertical lines to ensure that each is consistent and promotes good design.

Allan Kelly
allan@allankelly.net
http://www.allankelly.net

Acknowledgements

This pattern was the result of a conversation on the accu-general mailing list entitled: "overload 49 and state" with significant contributions from Kevlin Henney and Josh Walker, running from 18th June 2002. I am grateful to Kevlin for acting as initial pattern shepherd and Josh for reviewing the results and providing an additional example. The paper was further shepherded by Frank Buschmann in April 2003 for submission to EuroPlop. Again, I am most grateful to Frank for his time and interest.

In addition, I am most grateful to all in Workshop D at EuroPLoP 2003 for their many varied and useful comments concerning the pattern, their support and their suggestions for improvement.

Bibliography

Brown, J. B., Malveau, R.C., McCormick, H.W., and Mowbray, T.J. (1998) *Anti-Patterns*, Wiley.

Fowler, M. (2000) *Refactoring*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J, (1994) *Design Patterns*, Addison-Wesley.

Green, R. 2001 *How to Write Unmaintainable Code*, `http://www.web-hits.org/txt/codingunmaintainable.html`

Henney, K. 2002 *Objects for State*, `http://www.curbralan.com`

Jones, C. B. (1986) *Systematic Software Development using VDM.*

Liskov, B. (1988) "Data abstraction and hierarchy", *SIGPLAN Notices*, 23, 17-34.

Martin, R. C. (1996) "The Liskov Substitution Principle", *The C++ Report*, `http://www.objectmentor.com/resources/articles/lsp.pdf`.

Nobel, J. 1997, "Arguments and Results", *Pattern Languages of Programming (PLoP) conference*, Washington University, `http://citeseer.nj.nec.com/107777.html`

Patow, G., and Lyardet, F. (2003) "Open Arguments", *EuroPLoP 2003*, proceedings pending publication.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000) "Monitor Object", in *Pattern-Oriented Software Architecture 3*, Wiley, pp. 399-422.

Wordsworth, J. B. (1992) *Software Development with Z*.

Principles and Patterns Glossary

Pattern Name	Description
Arguments Object (Nobel, 1997)	"Large protocols [interfaces] are easy to use because they offer a large amount of behaviour to their clients. Unfortunately, they are often difficult or time consuming to implement, and for client programmers to learn. [...] Therefore: make an arguments object to capture the common parts of the protocol."
Blob (Brown, 1998, p.73)	"The *Blob* is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes, [...] Architectures with the *Blob* have separated process from data; in other words they are procedural-style rather than object oriented architectures."
Chain of Responsibility (Gamma, 1994, p.223)	"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."
Command (Gamma, 1994, p. 233)	"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."
Hide Forbidden Globals (Green, 2001)	"Since global variables are 'evil', define a structure to hold all the things you'd put in globals. Call it something clever like `EverythingYoullEverNeed`. Make all functions take a pointer to this structure (call it `handle` to confuse things more). This gives the impression that you're not using global variables, you're accessing everything through a "handle". "
Introduce Parameter Object (Fowler, 2000, p.295)	"Often you see a particular group of parameters that tend to be passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced with an object that carried all the data. It is worthwhile to turn these parameters into objects and just to group the data together. This refactoring is useful because it reduces long parameter lists, and long parameter lists are hard to understand."
Liskov Substitution Principle (Liskov, 1988)	"Functions that use pointers or references to base classes must be able to use objects of derived classed without knowing it." (Martin, 1996) When using class hierarchies as a means of data abstraction, sub-types must be able to fully substitute for the super-types.
Monitor Object (Schmidt, 2000, p.399)	Synchronises concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences.
Observer (Gamma, 1994, p.293)	"Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically."
Objects for State (Henney, 2002)	"Allow an object to alter its behaviour significantly by delegating state-based behaviour to a separate object."
Open Arguments (Patow, 2003)	"*Open Arguments* is used to create a generic interface for parameter passing, decoupling the API declaration of the procedures and functions from the type and number of the parameters they receive. A parameter block is passed from function to function, the block contains a dynamic store (often a map) of parameter names and values."
Singleton (Gamma, 1994, p.127)	"Ensure a class only has one instance, and provide a global point of access to it."

Microsoft Visual C++ and Win32 Structured Exception Handling

by Roger Orr

In an earlier article [1] I described some performance measurements when using exceptions in various languages on Windows.

A couple of people since then have asked me questions about how the windows exception model actually works and how it is used to implement `try...catch` constructs in MSVC.

That's quite a big question to answer, and this article is a start. There is quite a lot written about how to use these language features safely; but much less written about how they are implemented. There are several good reasons for this:

- lessons learned about the language features themselves apply to all versions of standard C++, whatever the platform, whereas details of the implementation are specific to both the vendor and the platform.
- knowing how it works is not necessary to using it
- the details are very sketchily documented and not guaranteed by Microsoft to remain unchanged

On the other hand I for one like to know what is going on "under the covers" so that:

- I can satisfy my 'how do they do that?' curiosity
- I can understand the flow of control when trying to debug application problems
- I can perhaps provide some platform specific value-added services.

In order to give some motivation to the investigation here's my task: I want to develop an 'exception helper' so I can print out a simple call stack for a caught C++ exception. Java and C# both let you print the stack trace for the exception, but standard C++ does not provide a way to do this. Although I understand why this feature is not part of the language I do miss it in C++ and would like to do what I can towards providing it for a specific platform, in this case MSVC.

There are some, rather intrusive, source level solutions involving adding code to the constructors of all exception types used in your application or adding code to each use of `throw`. This sort of solution means you must change the way exceptions are used throughout the code base which can be a large task even if you have access to all the source code, and impossible if not.

I'll describe writing a class for the MSVC compiler so that this code:

```
void testStackTrace() {
  ExceptionStackTrace helper;
  try {
    doSomethingWhichMightThrow();
  }
  catch(std::exception & ex) {
    std::cerr << "Exception occurred: "
              << ex.what() << std::endl;
    helper.printStackTrace( std::cerr );
  }
}
```

prints out a stack trace for the exception:

```
Exception occurred: A sample error...
  Frame       Code address
  0x0012FE70  0x7C57E592 RaiseException+0x55
  0x0012FEB0  0x7C359AED CxxThrowException+0x34
  0x0012FF10  0x004013CA throwIt+0x4a at
                  teststacktrace.cpp(32)
  0x0012FF18  0x004013E8
                  doSomethingWhichMightThrow+0x8
                  at teststacktrace.cpp(37)
  0x0012FF58  0x0040142A testStackTrace+0x3a at
                  teststacktrace.cpp(45)
  0x0012FF60  0x004014A8 main+0x8 at
                  teststacktrace.cpp(57)
  0x0012FFC0  0x00404E87 mainCRTStartup+0x143 at
                  crtexe.c(398)
  0x0012FFF0  0x7C581AF6 OpenEventA+0x63d
```

Processing the Stack Trace

Microsoft provide a debugging library, `DbgHelp.dll`, which provides among other things functions to walk up the stack and print out the return addresses. A full description of `DbgHelp.dll` is outside the scope of this article – I refer you to Matt Peitrek's MSJ article [5] or John Robbins' book [6] if you want more details.

The `StackWalk()` function provided by `DbgHelp.dll` takes nine parameters, but the key ones are a `StackFrame` and a `ContextRecord`. The `StackFrame` is an in/out parameter used to contain data for successive stack frames and the `ContextRecord` contains the thread state in a platform dependent manner. (Different definitions of the structure are given in `WinNt.h` and the correct one is picked by the C++ preprocessor). The `ContextRecord` is technically optional, but contains enough information to initialise the `StackFrame` and also improve the reliability of the `StackWalk` function so it is preferable to require one.

So here is a function prototype for a simple stack trace routine implemented using the functionality of `DbgHelp.dll`:

```
void SimpleSymbolEngine::StackTrace(
                    CONTEXT *pContextRecord,
                    std::ostream & os);
```

The implementation of this function sets up `AddrPC`, `AddrFrame` and `AddrStack` in a `StackFrame` record from the `Eip`, `Ebp` and `Esp` registers in the context record and then calls `StackWalk` repeatedly until the stack walk is completed. Each frame address is printed, together with the return address. Two functions in the `DbgHelp` library (`SymGetSymFromAddr` and `SymGetLineFromAddr`) are called to get any symbolic information available for the return address. Note that even if you don't have debug symbols for your program (and DLLs) `DbgHelp` will try to provide information based on any exported names from in DLLs.

A context record can be obtained in a variety of ways. As it is simply a snapshot of the thread state it could be built up manually using inline assembler to populate the various fields from CPU registers – or more easily by using the `GetThreadContext` call. The operating system also uses them in various places when managing thread state and finally they also crop up in exception handling.

The main reason to write the `ExceptionHelper` class is to obtain the context record of the thread state when the exception occurred. We can then use this key piece of data to extract the stack trace. Let's look at Microsoft's implementation of `try`, `throw` and `catch` in Win32 C++ to see how it lets us build something to extract this information.

Structured Exception Handling

Microsoft integrated standard C++ exception handling with window's own exception handling model: so-called "structured

exception handling" or "SEH" and this section tries to give an overview of what is happening inside SEH from an application's viewpoint – however you don't need to completely understand the principles to follow the `ExceptionHelper` code.

The definitive article about Win32 structured exception handling is by Matt Peitrek [2] and I refer interested readers there. However, his article focuses on the Microsoft extensions to support SEH: `_try`, `_except` and `_finally` and less on the language native concepts embodied in `try`, `throw`, etc. Other articles, such as [3], focus on what is happening at the assembler level which is great for the minority of programmers who understand assembler but not for the rest.

There is a relatively natural fit between the SEH exception model and the `try...catch` exception model in programming languages such as C++ so it is not too surprising that Microsoft decided to use this operating system level structured exception handling to provide the basis for their C++ exception handling code. However other implementors of C++ on the Win32 platform have not necessarily followed the same pattern.

Windows provides a portable exception architecture which recognises two main type of exceptions: 'system' exceptions such as an access violation or an integer divide by zero, which are also known as 'asynchronous' exceptions, and 'user' exceptions generated by a call to `RaiseException()`, which are also known as 'synchronous' exceptions. Each thread contains a linked list of exception handlers and when an exception occurs information about the exception and a context record for the thread are passed to each exception handler in the chain in turn for possible processing. There are several things each handler can do; the commonest cases are:

- return 'keep looking' (and the next exception handler will be called)
- unwind the thread context back to a known state and execute a failure path (in C++, a `catch` block).

If the context is to be unwound then each exception handler which is unwound off the stack is called so it can perform any required tidy-up. If all of the exception handlers return 'keep looking' the operating system has a final, process wide, exception handler which by default produces one of the 'Application Error' popups.

(Note that this is a slight simplification of the full picture)

Each handler has a signature like this:

```
DWORD exceptionHandler(
            EXCEPTION_RECORD *pException,
            EXCEPTION_REGISTRATION_RECORD
                        *pRegistrationRecord,
            CONTEXT *pContext);
```

Where:

- `pException` contains information about the exception being processed, such as the exception code and the fault address.
- `pRegistrationRecord` points to the current node in the list of exception handlers
- `pContext` contains the processor-specific thread state when the exception occurred

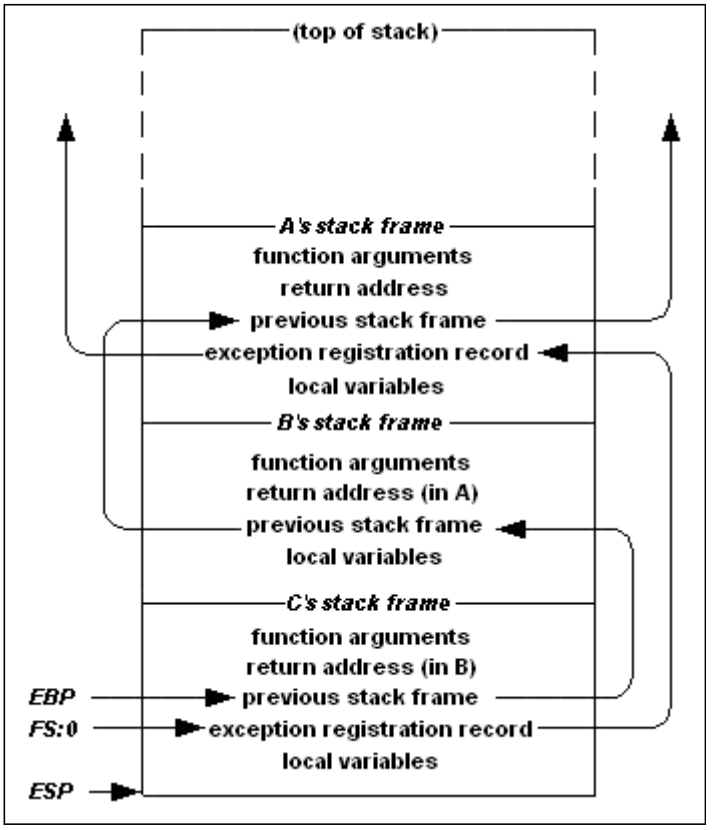
Our task is to retain the thread context from the last parameter so we can use it later in a call to the `StackTrace` function.

What makes this exception style 'structured' is that the chain of exception handlers exists in the thread's own stack. In a typically block-structured programming language each call to a function, method or procedure pushes a new activation frame onto the stack; this frame contains the current arguments, any local variables and the exception handler for this function (if any). Additionally, the algorithm which passes the exception along the chain of handlers naturally moves from the most recently called function up the stack to the top-most function.

In the Win32 world the `ESP` register contains the current stack pointer, by convention the current frame pointer is usually held in the `EBP` register and the `FS` selector register holds the base of the thread information block (TIB) which holds, among other things, the head of the exception chain.

To try and make this clearer here is a schematic representation of the bottom of the stack when function `A` has called function `B` which in turn has called function `C`.

Functions `A` and `C` have an SEH handler, but function `B` doesn't.



Inside the each stack frame the function arguments are above the frame register (and appear in assembler as [`EBP` + offset]) and local variables are below the frame register (and appear in assembler as [`EBP` – offset]). In practice things are more complicated than this – particularly when the optimiser gets involved – and the frame register `EBP` can get used for other purposes. To reduce the complexity of this article I'm not going to worry about optimised code.

We need insert our own exception helper object into the chain of exception registration records so we can extract the context record for the thrown exception.

MSVC Exception Handling

Before we can write our own exception handler we need to know a bit about how MSVC makes use of SEH handling to implement C++ exceptions. I've annotated the following code fragment with some of the key places that SEH handling is involved.

```
void thrower() {
  SomeClass anObject;  // 5
  throw std::runtime_error("An error"); // 2
}
void catcher() {  // 1
  std::string functionName("catcher");
  try {
    thrower();
  }
  catch(std::exception & ex) {  // 3
    std::cerr << functionName << ": "
              << ex.what() << std::endl;
  }
} // 4
```

1) When you write a function containing `try...catch` the Microsoft compiler adds code to the function prolog to register a structured exception handler for this function at the head of the thread's exception handler chain. The actual structure created for the exception registration extends the basic `EXCEPTION_REGISTRATION_RECORD`; the additional fields are used for managing the exception handling state and recovering the stack pointer.
2) `Throw` is implemented by calling `RaiseException` with a special exception code `0xe06d7363` (the low bits spell 'msc' in ascii). Other fields in the exception record are set up to hold the address and run-time type of the thrown object – in this case a `std::runtime_error` object.
3) The `catch` code is actually implemented by the exception handler. If the exception being handled has the special exception code value then the run-time type information is extracted and compared to the type of the argument in the `catch` statement. If a match is found (or a conversion is possible) then the exception chain is unwound and the body of the `catch` is entered, after which the execution will continue directly after the `try...catch` block. If a match is not found the exception handler returns the 'keep looking' value and the new handler in the chain will be tried.
4) On function exit the exception handler for `catcher` is removed from the chain.
5) There's another place that SEH code is needed of course – the destructor for `anObject` must be called during the unwind back to the `catch` statement.

   So there is actually yet another exception handler registered for `thrower` too, to deal with ensuring that `anObject` gets deleted. This one never tries to handle the exception but simply ensures local variables are destructed during the unwind.

One key thing about the way MSVC exception handling works is that it involves making extra calls down the stack. At point (2) the C++ runtime calls `RaiseException`, which snapshots the exception and thread state and then it in turn calls the code to work along the exception chain calling exception handlers. At point (3) when the exception handler for `catcher` gets control it is a long way down the stack. The exception chain is unwound by yet another call, this time to `RtlUnwind`. This function throws another exception along the exception chain with a special flag value `EXCEPTION_UNWINDING` set, giving each exception handler in turn a chance to do tidying up before it is removed from the exception chain. After returning from `RtlUnwind` the body of the `catch` statement is then called. When the catch body completes control returns back to the C++ runtime which completes tidying up the stack pointer, deletes the exception object and then resumes execution at the next instruction after the `catch` block.

So how does the `catch` block make use of the local variable `functionName` if it is so far down the stack when it gets control?

What the C++ runtime does is to use the extended exception registration record (passed to the handler as the second argument) to recover the value of the frame pointer `EBP`. Having reset the frame pointer the code in the `catch` body can make use of local variables and function arguments without difficulty. It is does not affect the function that the stack pointer is not simply a few bytes below the frame pointer but several hundred bytes below it.

The upshot is that, when the `catch` body is executed, the complete stack down to the location of the `throw` is still available in memory. The raw stack pointer will only be reset when the body of the stack completes, and before this point the call stack will not be touched. So if we can obtain the address of the context record that was passed into each exception handler as the third argument, the pointer will still be valid inside the body of the `catch`.

Looking back to the way the chain of exception handlers is processed we can see that if we can hook our code into the exception chain just before the compiler written exception handler we can extract information from the context record and then use that information inside the `catch` handler to allow us print a stack trace. Let's look at how we can do this.

Adding to the Exception Chain

The exception chain in Win32 consists of a singly linked list of `EXCEPTION_REGISTRATION_RECORD`s on the stack. Unfortunately Microsoft do not provide a C++ definition for this structure (possibly because it is different on each hardware platform running Windows) but they do provide one in an assembler include file `EXSUP.INC` which can be translated into C++ like this:

```
/** Typedef for the exception handler function
    prototype */
typedef DWORD (fnExceptionHandler)
      (EXCEPTION_RECORD *pException,
       struct _EXCEPTION_REGISTRATION_RECORD
                    *pRegistrationRecord,
       CONTEXT *pContext );
/** Definition of 'raw' WinNt exception
    registration record – this ought to be in
    WinNt.h */
struct _EXCEPTION_REGISTRATION_RECORD {
  struct _EXCEPTION_REGISTRATION_RECORD
         *PrevExceptionRegistrationRecord;
         // Chain to previous record
  fnExceptionHandler *ExceptionHandler;
         // Handler function being registered
};
```

So all we need to do, it seems, is to create an `_EXCEPTION_REGISTRATION_RECORD`, point `ExceptionHandler` to our exception handling function and insert the record at the top of the exception chain. Almost. There are some complexities with registering your own exception handlers.

Firstly, the code which walks the exception chain requires (on some but not all versions of Windows) that the nodes in the chain are registered in strict address order. Fortunately the compiler always puts local variables below the exception registration record so by using a local variable for our exception helper we should

always be able to insert it into the chain before the compiler generated exception registration record.

Secondly, I want to register the exception handler in a constructor. This function too has a compiler-generated exception handler. I must ensure that the handler is registered in the chain above the record for the constructor or my exception handler will be unregistered when the constructor completes! Additionally I want the code to work properly should there be two or more exception helper objects in a single function, and it is not in general possible to fix the offsets in the stack frame assigned by the compiler for local variables.

Lastly, as part of the security improvements included with Visual Studio .NET 2003 and Windows Server 2003/Windows XP service pack 2, the exception handling function to be called must be marked with a special attribute (SAFESEH) at link time so it will appear in the "Safe Exception Handler Table" in the load configuration record. Failure to do this results in a security exception occurring at runtime which usually terminates the process. This check has been added to Windows to prevent security exploits that use buffer overrun in order to replace the exception handler address on the stack with a pointer to injected code. The SAFESEH attribute can only be granted by assembler code so it is therefore necessary, when using Visual Studio .NET 2003, to make use of a very simple piece of assembler code to add this attribute to the exception handling function.

Note: the assembler `ml.exe` provided with the first Beta edition of Visual Studio 2005 access violates when using `/safeseh` [4] and that from 2003 must be used.

One mechanism I've found that provides good ease of use under the above constraints is to create a common base class for my own exception handlers. This class contains a static exception handling function which can be marked, once and for all, with the SAFESEH attribute. This common handler then makes a virtual call into the derived class for the specific action required for exception handling.

```
class ExceptionHelperBase : public
_EXCEPTION_REGISTRATION_RECORD {
public:
  /** Construct helper object */
  ExceptionHelperBase();
  /** Make safe to extend */
  virtual ~ExceptionHelperBase() {}
  /** Allow subclass to hook exception */
  virtual void onException(
                EXCEPTION_RECORD *pException,
                CONTEXT *pContext) = 0;
private:
  // Disable copy and assign
  ExceptionHelperBase(
                ExceptionHelperBase const &);
  ExceptionHelperBase& operator=(
                ExceptionHelperBase const &);
  // The one and only exception handler function
  static fnExceptionHandler exceptionHandler;
};
```

The exception handling function simply casts the exception registration record back to an `ExceptionHelperBase` and invokes `onException`:

```
DWORD ExceptionHelperBase::exceptionHandler(
      EXCEPTION_RECORD *pException,
      struct _EXCEPTION_REGISTRATION_RECORD
                        *pRegistrationRecord,
      CONTEXT *pContext) {
  ExceptionHelperBase &self =
      static_cast<ExceptionHelperBase&>(
                        *pRegistrationRecord);
  self.onException(pException, pContext);
  return ExceptionContinueSearch;
}
```

I would like my exception class to register itself in the constructor and deregister itself in the destructor. Unfortunately I can't simply do this in the `ExceptionHelperBase` constructor and destructor without risking problems if I get an exception during the constructor/destructor code itself.

However, a use of the 'curiously recurring template pattern' fixes this problem and ensures registration happens last in the constructor and first in the destructor:

```
template <typename RegistrationRecord>
class AutoRegister : public RegistrationRecord {
public:
  /** Auto-register an exception record for
      'RegistrationRecord' */
  AutoRegister() : RegistrationRecord() {
    registerHandler(this);
  }
  /** Unregister and destroy an exception
      record */
  ~AutoRegister() {
    unregisterHandler(this);
  }
};
```

Where `registerHandler` will install the handler in the exception chain and `unregisterHandler` will remove it from the chain by using standard logic for singly-linked lists. The list head is held in the `NT_TIB` structure pointed to by the FS register and the list tail is the value `-1`.

Processing the Exception

The first derived class simply prints out the exception information to demonstrate that things are working properly:

```
class ExceptionHelperImpl1
            : public ExceptionHelperBase {
  /** Print the address of the exception
      records */
  virtual void onException(
              EXCEPTION_RECORD *pException,
              CONTEXT *pContext) {
    printf("pException: %p (code: %p,
              flags: %x), pContext: %p\n",
           pException,
           pException->ExceptionCode,
           pException->ExceptionFlags,
           pContext);
  }
};

typedef AutoRegister<ExceptionHelperImpl1>
                              ExceptionHelper1;
```

Since this code is executing while an exception is actually being processed I used `printf()` rather than `std::cout` to avoid any potentially harmful interactions with the standard library.

Sample code:

```
int main() {
  ExceptionHelper1 helper;
  try {
    printf("About to throw\n");
    throw std::runtime_error(
                          "basic exception");
  }
  catch(std::exception & /*ex*/) {
    printf("In catch handler\n");
  }
  return 0;
}
```

When executed this program generates output for two exceptions:

```
About to throw
pException: 0012FBA0 (code: E06D7363, flags:
                        1), pContext: 0012FBC0
pException: 0012FBA0 (code: E06D7363, flags:
                        3), pContext: 0012F670
In catch handler
```

The second call is generated by the Microsoft supplied exception handler unwinding the exception chain. This is easily identified as the `EXCEPTION_UNWINDING` flag (value `2`) is set in `pException->ExceptionFlags` for the second exception. For our purposes we want to extract context data from the first call since this context describes the thread state when `throw` was executed. (Note that our exception handler is removed from the chain during the exception unwind so would need to be re-inserted to catch subsequent exceptions in the same scope)

We now have everything we need for the basic version of the code to print a stack trace:

```
class ExceptionStackTraceImpl
              : public ExceptionHelperBase {
public:
  ExceptionStackTraceImpl()
                        : pSavedContext(0) {}
  /** Use the saved pointer to print the stack
      trace */
  void printStackTrace(std::ostream & os)
                                      const {
    if(pSavedContext != 0)
      SimpleSymbolEngine::instance()
              .StackTrace(pSavedContext, os);
  }
private:
  /** Capture the thread context when the
      initial exception occurred */
  virtual void onException(
                EXCEPTION_RECORD *pException,
                CONTEXT *pContext) {
    if((pException->ExceptionFlags
             & EXCEPTION_UNWINDING) == 0) {
      pSavedContext = pContext;
    }
  }
  PCONTEXT pSavedContext;
     // context record from the last exception
};
typedef AutoRegister<ExceptionStackTraceImpl>
                        ExceptionStackTrace;
```

We have now achieved the original aim of be able to print a stack trace in the catch block.

Interaction With Normal SEH

This method of 'hooking' in to the MSVC handling of C++ exceptions means that the exception handler is also called for every other SEH exception, such as access violation. In released versions of MSVC the implementation of `catch(...)` also processed these types of exceptions. Although this seems at first sight to be a good thing it actually tends to cause more problems than it solves.

One particular issue is that genuine problems such as corrupt memory, I/O errors on the paging file or load time problems with DLLs get handled in the same way as a C++ exception of unknown type, by code not written to deal with these error conditions. For current versions of MSVC it is usually best to avoid use of `catch(...)` unless either the code re-throws the exception or terminates the process.

Visual Studio 2005 Beta 1 handles non-C++ SEH exceptions in a `catch(...)` only when the compiler flag `/EHa` is set, which is a great improvement and gives maximum flexibility.

Whether or not `/EHa` is specified we can use `ExceptionHelper` to extract information about the OS exception. Microsoft provide some other ways to achieve a similar end, `__try`/`__except` and `_set_se_translator`, but they are not total solutions. Also not all compler vendors provide such extensions and the `ExceptionHelper` code could still be used to extract information about the exception.

For example I modified `ExceptionHelper1` class for `gcc` on win32 (a couple of minor changes were required for a clean compile). Since `gcc` does not seem to use SEH for C++ exception handling `ExceptionHelper1` did not capture information for such exceptions but it did do so for Win32 exceptions, such as access violations. For a "proof of concept" I changed the exception handler to throw a `std::runtime` exception rather than printing the exception information and was successful in mapping an SEH exception into a C++ exception, thus allowing additional tidyup to be performed before the program exited.

Conclusion

I have given a brief overview of how the Microsoft compilers on Win32 implement C++ exception handling. Using this information we've seen a simple class which enables additional information to be obtained about the exception during program execution.

What is the main strength and weakness of this approach?

On the positive side it enables better diagnostic information to be produced at runtime for MSVC on Win32. This can significantly reduce the cost of finding bugs, since enough information might be gathered in the field to identify the root cause. Without this extra information it might be necessary to try and reproduce the problem under a debugger, with potential difficultly of getting the right execution environment to enure the problem does in fact appear.

The main weakness is that the code is platform specific and relies on undocumented behaviour of the compiler. Other compilers under Win32 do not use the SEH mechanism to handle C++ exceptions so this code is useless should your code need to be portable to them. The implementation of the exception mechanism even under 64 bit Windows is not the same as for 32 bit Windows, so the technique described here will not work unchanged (if at all) in that environment even for Microsoft compilers.

[concluded at foot of next page]

A Mini-project to Decode a Mini-language – Part One

by Thomas Guest

This article appears in two parts. Part One – this part – describes the first stages of a mini-project to write a codec for a mini-language. (If you don't know what codecs and mini-languages are, don't worry. Read on!) Part two will describe the later stages of this project and present an actual implementation of the codec.

Keen readers will, then, have the opportunity to try out an implementation of their own before part two appears, based on the specification arrived at during the course of the following paragraphs. As encouragement, I provide a (link to a) data suite which can be used for test purposes.

Motivation

The motivation for this article comes from "The Art of UNIX Programming" by Eric Raymond ([Raymond]). This is one of the most inspiring books I've read on how to write software: although firmly rooted in the traditions of the UNIX operating system, the culture and philosophy it describes applies far more widely. It has reinforced my belief that software development can indeed be an art.

Having read this book, I wanted to put some of its ideas into practice. So I set myself a mini-project.

An Idea For a Mini-project

As a starting point, I'd like to summarise two of [Raymond]'s most important lessons.

- Data structures, not algorithms, are central to programming.
- Prefer text file formats: they're human-readable and extensible. If you must use a binary format then invest in a tool which converts from this format to a textual one and back again. This will facilitate working with your data.

I work in a domain where the need for compression requires the use of binary file formats: namely digital television (DTV). Digital video is typically MPEG-2 encoded. This is a highly compressed encoding designed to squeeze as much content as possible into a limited bandwidth. MPEG-2 encoding also allows video and audio content to be combined with metadata: for example, a television programme might be accompanied by a text description of itself and of what's showing next[1].

1 The proper term for these particular items of metadata is Event Information, present and following. Since this article is not primarily about digital television I shall avoid such jargon as far as possible.

To get to grips with this metadata a conversion tool is required. The specific task I set myself, then, was to write a tool to convert MPEG-2 metadata from binary to text, and, if required, to reverse the process. Such an encode/decode tool is commonly referred to as a 'codec'.

My project, then, was to implement a digital television codec.

The MPEG-2 Bit Stream Syntax: An Example of a Mini-language

The metadata format is specified using the MPEG-2 bit stream syntax which is defined in [ISO/IEC 13818-1]. Data items are described by name and length in bits using a C-like procedural syntax. An example makes this clear:

```
TS_program_map_section() {
  table_id                  8
  section_syntax_indicator  1
  '0'                       1
  reserved                  2
  section_length           12
  program_number           16
  reserved                  2
  version_number            5
  current_next_indicator    1
  section_number            8
  last_section_number       8
  reserved                  3
  PCR_PID                  13
  reserved                  4
  program_info_length      12
  for(i=0; i<N; i++) {
    descriptor()
  }
  for(i=0;i<N1;i++) {
    stream_type             8
    reserved                3
    elementary_PID         13
    reserved                4
    ES_info_length         12
    for(i=0; i<N2; i++) {
      descriptor()
    }
  }
  CRC_32                   32
}
```

[continued from previous page]

The decision depends on the relative balance between these two items. However, having isolated the logic into a single class `ExceptionStackTrace` it can be conditionally compiled to a do-nothing implementation on other platforms, or if may even be possible to re-implement the logic for another platform.

Roger Orr
rogero@howzatt.demon.co.uk

References

[1] Roger Orr, "Efficient Exceptions?", *Overload 61*, June 2004

[2] Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, http://www.microsoft.com/msj/0197/Exception/Exception.aspx

[3] Jeremy Gordon, *Win32 Exception handling for assembler programmers*, http://www.jorgon.freeserve.co.uk/Except/Except.htm

[4] MSDN Product Feedback Center, http://lab.msdn.microsoft.com/ProductFeedback, search on keyword "FDBK12741"

[5] Matt Pietrek, *Improved Error Reporting with DBGHELP 5.1 APIs*, http://msdn.microsoft.com/msdnmag/issues/02/03/hood/default.aspx

[6] John Robbins, *Debugging Applications for Microsoft .NET and Microsoft Windows*, Microsoft Press

What we have here is the bit stream syntax for a section of the Program Map Table. The first 8 bits give the table id (which happens to be 2, for this particular table); the single bit which follows provides the section syntax indicator; the next bit is always set to zero; the next two bits are reserved (and should each be set to one); and so on, until we get to the 32 bit CRC[2].

To provide a little context: the Program Map Table (PMT) supplies basic information about the digital television services present in an MPEG-2 transport stream. A section of this table – as shown above – defines the elementary streams which comprise a single television service. For example, the PMT for the BBC1 digital television service consists of a video stream, an audio stream, a subtitle stream and some data streams. Of course, [ISO/IEC 13818-1] defines the format of many other tables and sections, and related specifications – such as [EN 300468] define many more.

This textual specification of a binary format is an example of what [Raymond] terms a mini-language. In fact we have a Turing-complete mini-language: that is, it allows `for` loops and conditionals. The particular example shown here does not include any conditionals, though we do have nested loops. Note also the referenced `descriptor()` items. To fully parse the `TS_program_map_section()` we'll need the `descriptor()` format specified too:

```
descriptor() {
  descriptor_tag       8
  descriptor_length    8
  for(i=0; i<N; i++) {
    private_data_byte  8
  }
}
```

Complications

The syntax is easy to read, particularly to anyone familiar with C. However, if we look more closely at the `for`-loop in Example 2, although it's apparent that `i` must be an integral loop counter, it's less clear where `N` is defined. Similarly, in Example 1, how do we find the values of `N1` and `N2`?

[ISO/IEC 13818-1] answers these questions. In Example 2, the `descriptor_length` data element encodes an unsigned integer which tells us how many bytes of data are to follow: so `N` is simply the value obtained by decoding `descriptor_length`. Example 1 is not quite so simple. `N` is easy enough – it's as many variable-length descriptors as it takes to fill the total length specified by `program_info_length`. `N2` is similarly the number of descriptors to fill the length specified by the most recent occurrence of `ES_info_length`. For `N1` however, we have to keep decoding elementary streams until the following is true:

```
Sum of elementary stream lengths (in bytes)
   == 'section_length' -
      - (2 + 5 + 1 + 8 + 8
         + 3 + 13 + 4 + 12) / 8
      - 'program_info_length'
      - 32 / 8
```

We have to divide by 8 since field widths of values within the PMT section are given in bits – but length fields give values in bytes.

2 I assume the 'reserved' parts of the section are included to provide room for a degree of future extensibility. But not much room. This is one of the reasons why [Raymond] advocates text file formats: *"if you need a larger value in a text format, just write it."*

Despite these complications, the encoding is well-designed: we can parse these binary data structures sequentially without needing to look ahead; and we can skip over any bits we're not interested in.

In fact, the more closely we inspect our examples, the more we notice. This is good. Recall that data structures, not algorithms, are central to programming. Already we're getting stuck into the data.

While we're in this positive frame of mind, let's review the full range of control structures required by the [ISO/IEC 13818-1] bitstream syntax:

```
while(condition) {
  data_element
  ...
}

do {
  data_element
  ...
}
while(condition)

if(condition) {
  data_element
  ...
}
else {
  data_element
  ...
}

for(i=0;i<n;i++) {
  data_element
  ...
}
```

So, we've got pretty much C's control structures, excepting `switch`, `break`, `return` and `continue`.

This is starting to look alarming. How complex can a `condition` be? How shall we handle three different looping constructs? Our mini-project has become rather bigger than we imagined.

Back to the Data

The thing to do at this point is to shelve these concerns and get back to specifics. So, I got hold of some PMT section data and parsed it by hand. I used two types of data:

- PMT sections pulled out of recorded digital TV broadcasts
- a simple PMT section synthesised by hand.

I shall spare you the details. Note though that in parsing by hand we're already starting work on our text output format. For example, given the binary contents of a synthesised descriptor:

```
0a 04 0a 0b 0a 0b
```

and recalling the descriptor syntax:

```
descriptor() {
  descriptor_tag        8
  descriptor_length     8
  for (i=0; i<N; i++) {
    private_data_byte   8
  }
}
```

a suitable output might be:

```
descriptor() {
  descriptor_tag      8 = 0x0a
  descriptor_length   8 = 0x04
  for (i=0; i<N; i++) {
    private_data_byte 8 = 0x0a
    private_data_byte 8 = 0x0b
    private_data_byte 8 = 0x0a
    private_data_byte 8 = 0x0b
  }
}
```

I have deliberately chosen an output format which closely resembles the syntax definition. The loop has been unrolled, but I have retained the loop control to indicate the structure and origin of the data. I have chosen a hexadecimal representation for the data values – always a good choice for binary data – and explicitly indicate the numeric base used by prefixing these values with the string `0x`. Finally, I have retained the bit widths for convenience: this will mean that when converting from text to binary, there will be no need to refer back to the descriptor syntax.

Referrring back to our motivating reference, we see we have instinctively followed one of [Raymond]'s recommendations:

"when filtering, never throw away information you don't need to".

(Here, the term "filter" is used in its Unix sense, and applies well to a codec; and the reasoning is that any discarded information can never be used in any program further down the Unix pipeline). In our example, we can see that the output includes all the information carried by both the descriptor syntax definition and by the example descriptor.

Handling Failures

Suppose our descriptor was too short:

```
0a 04 0a 0b 0a
```

What should our codec make of such data?

Maybe something like this:

```
descriptor() {
  descriptor_tag      8 = 0x0a
  descriptor_length   8 = 0x04
  for (i=0; i<N; i++) {
    private_data_byte 8 = 0x0a
    private_data_byte 8 = 0x0b
    private_data_byte 8 = 0x0a
  >>> ERROR: end of data reached. descriptor()
                                      incomplete
```

It's perhaps premature to tie down how errors should be handled, other than to say that they should draw attention to themselves, that they shouldn't crash the codec, and that they should provide useful diagnostics. But it certainly isn't premature to include some malformed data in our test set.

Another point [Raymond] makes about data conversion tools is that they should be generous in what they accept (as input) but rigorous in what they emit (as output). In our case, this means that a user might change the layout of a text version of a `descriptor()` to read like this:

```
descriptor()
    {   descriptor_tag 8 = 0xA
        descriptor_length 8 = 0x4
        for (i = 0; i < N; i++)
        {    private_data_byte 8 = 0xA
             private_data_byte 8 = 0xB
             private_data_byte 8 = 0xA
             private_data_byte 8 = 0xB
        }
    }
```

and still expect the codec to convert this to binary as:

```
0a 04 0a 0b 0a 0b
```

One of the great benefits of having a codec is that we can generate binary data from an easy-to-edit textual form: it would be a severe limitation if the encoding process was over-sensitive about whitespace, layout, or the capitalisation of hexadecimal numbers.

Reducing Project Scope

Whilst tinkering around with my test data set, I've also been paging through the MPEG-2 bitstream syntax. The bad news is that the expressions which appear in conditionals may be quite complex, making use of all the usual C arithmetic, bitwise, logical and relational operators as well as a few domain-specific additions.

The good news is that we can make good progress if we restrict our scope as follows:

- Restriction 1: restrict the control structures to `for() {...}` and `if (condition) {...} else {...}`
- Restriction 2: restrict conditions to the form `field == value`

These restrictions will not make a lot of sense to end users. In end user terms, we can aim for a first release of our codec which will only support sections from the following four tables:

- Program Association Table (PAT)
- Conditional Access Table (CAT)
- Network Information Table (NIT)
- Program Map Table (PMT)

This reduced scope may seem rather limiting.

Note however that these four tables – collectively termed the Program Specific Information (PSI) Tables – *"contain the necessary and sufficient information to demultiplex and present programs"* ([ISO 13818-1]).

Note further that our syntactic restrictions will not stop us from extending our codec to handle the complete set of DVB Service Information (SI) tables ([EN 300468]), which contain just about all of the metadata used in European digital broadcasts. Note finally that we remain faithful to our aims: [Raymond] emphasises the need to release early and often, so that users can drive (and implement, even, in an open source world) future developments. By reducing scope, we allow for this early feedback. We must take care, though, to follow another Unix maxim, and keep our design extensible.

A Prototype Descriptor Decoder

```
/**
 * @brief Decode a descriptor.
 * @param begin The start of the descriptor data
 * @param end One past the end of the descriptor
 *            descriptor data
 * @param out Output stream for the decoded data
 */
bool decodeDescriptor(desc_iter begin,
         desc_iter end, std::ostream & out) {
  out << "descriptor() {\n";
  if(begin != end) {
    out << "    descriptor_tag 8 = "
        << *begin++ << "\n";
  }

  // We don't know yet how much data we need to
  // decode.  Use a special non-zero value to
  // indicate this.
  unsigned int to_decode = 0xff;

  if(begin != end) {
    to_decode = *begin++;
    out << "    descriptor_length 8 = "
        << to_decode << "\n";
    out << "    for(i=0; i<N; i++) {\n";
    while(begin != end && to_decode != 0) {
      out << "        "
          << "private_data_byte 8 = "
          << *begin++ << "\n";
      --to_decode;
    }
    out << "    }\n";
  }
  if(begin != end) {
    out << "ERROR: descriptor() too long.\n";
  }
  else if (to_decode != 0) {
    out << "ERROR: end of data reached. "
        << "descriptor() incomplete.\n";
  }
  else {
    out << "}\n";
  }
  return begin == end && to_decode == 0;
}
```

The function above is a direct first attempt at writing a descriptor decoder. Whilst there may be some mileage in this approach, some weaknesses are apparent:

- The indentation is fixed. This won't do if we are decoding a descriptor in the broader context of a PMT section, when it can appear at two different levels.
- The error handling is clumsy.
- Data – in this case, the descriptor's syntax – has become muddled with control flow.

Now is not the time to deal with these weaknesses. We shall simply note that the first is simple to fix and the second can easily be improved on. It's the third weakness which, in the longer term, will lead to code which is harder to understand, maintain and extend. On the other hand, this function demonstrably works on our test data set, which is encouraging; and it's not hard to see how the approach taken could be extended to decode PAT, CAT, NIT and PMT sections – which is all we've decided to do.

Design Alternatives

We are now in a good position to consider the design of our dtv-codec. Three alternatives spring to mind:

- Implement a descriptor-codec, a pmt-codec, and so on as required. Here, each mini-codec understands its own designated part of the syntax. Then the dtv-codec simply farms out work as appropriate. This extends the direct approach described above.
- Implement a general dtv-codec which understands the full bitstream syntax and can use it to parse an arbitrary section format. All that then remains is to prime this codec with the required section formats.
- Devise a code generator which, given a section format, will generate a program to code/decode that particular format.

All three alternatives are good, and all seem in line with our motivating aims. The third, in particular, exemplifies [Raymond]'s *"Rule of Generation: Avoid hand-hacking; write programs to write programs when you can."*

In choosing which route to take, we should remember the [XP] mantra: *"Do the simplest thing possible"*; which, in UNIX-speak, becomes the more prosaic: *"Keep it Simple, Stupid!"*

More Details

For those who want to implement their own codec I have posted a zip archive to my website [HomePage]. This archive contains:

- binary PAT, CAT, NIT and PMT sections
- synthesised text sections
- alternative text versions
- malformed binary sections
- the relevant section syntax definitions
- table_id values
- a README

For those who'd prefer to see my attempt; you'll have to wait for part two of this article.

Conclusions

Progress has been made, and without the need to compromise our artistic aims. Even before we've completed the project, we've started to receive the main benefit: of understanding our data.

The second part of this article is where we compromise, get our hands get dirty, bite off more than we can chew – that is, we write some code.

Thomas Guest
thomas.guest@ntlworld.com

References

[Raymond] Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley 0-13-142901-9

[ISO/IEC 13818-1] *Information Technology - Generic Coding Of Moving Pictures And Associated Audio: Systems* Recommendation H.222.0 ISO/IEC 13818-1

[ETSI EN 300468] *Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems*

[XP] *Extreme Programming*, http://www.extremeprogramming.org

[HomePage] http://homepage.ntlworld.com/thomas.guest

Garbage Collection and Object Lifetime

by Ric Parkin

It seemed a simple bug report. “When we close the editing screen the framework asks the user to save the temporary portfolio we use internally as storage. Make sure it gets removed cleanly instead”. “Should be easy,” I thought. “We can’t be leaking the objects, as the whole form and its helpers are written in C#. I just need to destroy the portfolio object at the right time.”

The resulting solution opened my eyes to what Garbage Collection does and, more importantly, what it doesn’t do.

To understand this, we’ll go back to the very basics of memory and object management, and see what various techniques are available. I’ll concentrate on the C family of languages: C, C++, C# and the upcoming C++/CLI.

Memory and Object Lifecycle

The diagram below shows the stages in the life of memory and objects.

Raw memory is very simple: you acquire some from a pool of available memory, use it, and release it back to the pool to be reused. Failing to release it causes it to be considered “leaked”.

Objects are slightly more complex because as well as obtaining the raw memory for their storage, they need to be initialised to a usable state and establish their class invariant, and have that state destroyed before releasing the memory. If an object is not destroyed then its state is considered leaked, which is important if that state is a scarce non-memory resource such as system file handles.

Let’s look at some pseudo-code for creating and destroying an object, and then see how the C family languages map onto each part:

```
Memory_location memory_for_T
                = Acquire_Memory(size_of_T);
if(succeeded) {
  T_location T_object
                = Initialise_T(memory_for_T);
  if(not succeeded)
    Release_Memory(memory_for_T);
  else {
    // use T_object....
    Destroy_T(T_object);
    Release_Memory(memory_for_T);
  }
}
```

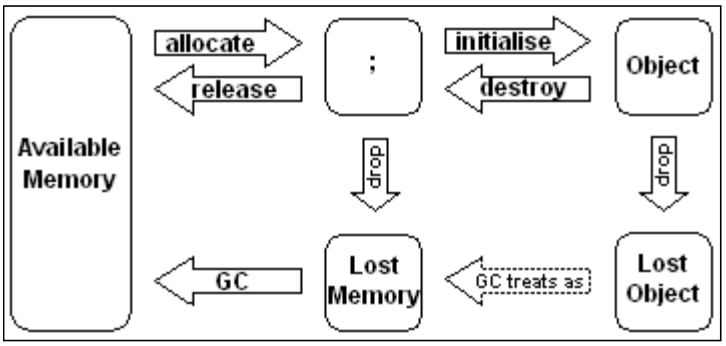


Figure 1: Memory and Object Lifecycle

There are four situations I’ll look at: creating an object on the stack; as a base class of some other object; as a member of an object; and on the heap. We’ll illustrate these by considering a class described loosely as:

```
class B : A {
  C c;
  D* d;
}
```

We’ll initialise each instance of `A`, `B`, `C`, and `D` with the numbers 1, 2, 3, and 4.

C++

C++ provides a very simple and clean solution

```
class B : public A {
public:
  B(int b_param)
     : A(1),
       c(3),
       d(new D(4)) {}
private:
  C c;
  std::auto_ptr<D> d;
};

int main() {
  B b(2);
  // use b
}
```

In C++, and all the other languages, the size of an object is worked out by the system, and takes into account all the space for the sub-objects.

If objects are constructed on the stack, then the memory is acquired and released automatically, often by just adjusting the stack pointer. Constructors play the part of the initialise function, and the programmer usually writes them, although there are cases where the compiler will generate them. When objects go out of scope the destructor is automatically called and the memory released.

Destructors are the destroy functions and the compiler automatically calls the destructors for base classes and members. It can even write them too – if there is nothing else needed other than the members’ destructors to be called, then the required destructor is trivial, and the compiler will generate it for us if we leave it out altogether.

Members of other objects are very similar to stack variables. In particular, when the outer object is destroyed, all its member objects are destroyed too.

Allocation on the heap is done using a “new-expression” which allocates memory and calls the required constructor. A “delete-expression” calls the destructor and frees the memory.

Experts at Kipling’s game of “Kim” may have spotted something missing – error checking. Fortunately the compiler generates it all for you using exceptions. I’d have to be more careful if I had raw pointers in my class, but wrapping them up in `auto_ptr` makes that problem go away and I can be lazy and correct, which is every programmer’s ideal.

C

C is more verbose – after all you have to do a lot more yourself. The only things the compiler will do for you is allocate and deallocate space on the stack and for `struct` members, and tell you the size needed for objects using the `sizeof` operator.

```
typedef struct B_tag {
  A a;
  C c;
  D* d;
} B;

D* D_new(int d_param) {
  void* memory = malloc(sizeof(D));
  if(memory == NULL)
    goto malloc_failed;

  D* d = D_init(memory, d_param);
  if(d == NULL)
    goto init_failed;

  return d;

init_failed:
  free(memory);

malloc_failed:
  return NULL;
}

B* B_init(void* memory, int b_param) {
  A* a = A_init(memory, 1);
  if(a == NULL)
    goto A_failed;

  C* c = C_init(memory + offsetof(B, c), 3);
  if (c == NULL)
    goto c_failed;

  d = D_new(4);
  if(d == NULL)
    goto d_failed;

  return (B*)memory;

d_failed:
   C_destroy(c);
c_failed:
   A_destroy(a);
A_failed:
   return NULL;
}

void* B_destroy(B* b) {
  // assume destruction can't fail
  free(D_destroy(b->d));
  C_destroy(&b->c);
  A_destroy((A*)b);
}

int main() {
  B b;
  B_init(&b, 2);
  // use b
  B_destroy(&b);
}
```

This is directly analogous to the C++ solution, and illustrates the sort of tricks the C++ compiler is doing behind the scenes, in particular the error checking and clearing up of partially constructed objects. But it is a lot of work. I'll leave it as an exercise for the reader to come up with a better solution in terms of writing and maintaining this sort of code.

What Is Garbage Collection?

Garbage Collection replaces manual releasing of memory such that 'leaked' memory is automatically reclaimed by the system and is then available for use.

It does this by finding objects that are no longer needed (technically, objects that are unreachable from "root" objects such as global variables and the stack by following member references) and reclaims their memory for future use. It can be compared to treating the program as having an infinite amount of memory – if you can never run out, then you don't need to bother to delete anything, and all objects can live forever and can be thought of as immortal [1].

It is very tempting, when starting to use garbage collection, to think that it means you don't have to worry any more about the tedious work of keeping track of object ownership and lifetimes, and the programmer can concentrate on more interesting and more productive work.

> *"... Garbage collection relieves you from the burden of freeing allocated memory ... First, it can make you more productive..." [2]*

> *"A second benefit of garbage collection ... is that relying on garbage collection to manage memory simplifies the interfaces between components ... that no longer need expose memory management details ("who is responsible for recycling this memory")." [3]*

Unfortunately, this misses a subtle point in the relationship between ownership, object lifetimes, and memory management – they aren't the same thing. Garbage Collection frees you from having to clean up the memory, true. But Ownership and Lifetime still have to be carefully considered as part of design.

For example, holding on to object references for too long, or giving them to global objects, will keep them locked into memory. This is often referred to as a memory leak, although it is achieved by incorrectly holding onto things for too long, and not by forgetting to clean up as in C++.

C# and .Net

In C# construction is very much like C++ in that the `new` keyword combines allocating the memory and calling the constructor.

There is no explicit memory deallocation stage – that's done automatically by the Garbage Collector – but is there something that can destroy an object? Not for releasing memory for its members – again that's done by the Collector – but something for cleaning up non-memory resources at a specific time?

There is a special function called when an object is being reclaimed – the finalizer. At first sight this looks very much like a destructor (the C# syntax is the same, and the designers of Managed C++ in VC7 thought they were the same – MC++ destructors are actually the finalizer in disguise), but it has since become clear that the finalizer can't be used to destroy an object, for three reasons:

- **You don't know when it gets called.** Things get finalized when the garbage collector runs, but all you know is that is may run at some unspecified point in the future, so you can't rely on it being called at a specific time
- **You don't know how many times it gets called, if at all.** It's possible that the program finishes before the collector runs, in which case the finalizer is never called. Also, an object that has been finalized can be kept alive using the `ReRegisterForFinalize` method, and then finalized again. And again. And again.
- **You can't do much in it.** When your finalizer is running, you don't know which other managed objects you have references to have already been finalized, so unless your design guarantees they're still living – in other words, you've carefully thought out their lifetimes – you can't touch any other objects. The only sensible thing you can do is to log some information somewhere to say it's been finalized.

It is sometimes recommended to use the finalizer to clean up important unmanaged resources that need to be released, such as handles from the operating system that the Garbage Collector doesn't know about. Unfortunately you may have run out of these resources before the collector runs and the finalizers get called, so you can't rely on that[1].

Dispose

Recall in my original problem, that I needed to tidy up a particular object at a particular time. A common solution is to write a teardown method, and the .Net designers have provided a standard interface: `IDisposable`, which has a single method `Dispose()` to be called when you want the object to clean up and "die". However, as there can be other references to the object, `Dispose` may be called on an object multiple times, and it is also allowed that a disposed object may be reused, for example a disposed File Object could be reopened, and become "alive" again, but I suggest that this would get too confusing to recommend – keep it simple: `Dispose` destroys the object, and nothing else can use it afterwards.

Used like this, `Dispose` is a candidate for the equivalent of a destructor. If an object has resources that must be released at a specific time, implement `Dispose` and remember to call it. C# has even added help to the language to do this – `using` – which will automatically call `Dispose` on its argument at the end of a statement block, in a similar way to `auto_ptr`, or Boost's `scoped_ptr`.

So finally, here's the example in C#

We'll have our base class `A` inherited from a helper class `Disposable` – it's based on a pattern for writing disposable objects where both the Finalizer and the `Dispose` methods are dispatched to a single virtual helper [4]. Some classes in the .Net framework such as `UserControl` use this technique

```
public class Disposable : IDisposable {
  private bool isDisposed = false;

  public Disposable() {}

  ~Disposable() {
    Dispose(false);
  }

  public sealed virtual void Dispose() {
    if(!isDisposed) {
      isDisposed = true;
      Dispose(true);
      GC.SuppressFinalize(this);
    }
  }

  protected virtual void Dispose(
                          bool isDisposing) {}

  protected sealed void TryDispose(
                          Object object) {
    TryDispose((IDispose)object);
  }

  protected sealed void TryDispose(
                          IDispose idispose) {
    if(idispose != null)
      idispose.Dispose();
  }
}

public class B : A {
  public B(int b_param)
     : base(1) {
    try {
      c = new C(3);
      d = new D(4)
    }
    catch(Exception e) {
      Dispose();
      throw e;
    }
  }

  public override void Dispose(
                          bool isDisposing) {
    if(isDisposing) {
      // dispose of managed resources here
      TryDispose(d); d = null;
      TryDispose(c); c = null;
    }
    // dispose of unmanaged resources here
    // and call the base class
    base.Dispose(isDisposing);
  }
```

1 As Java's Garbage Collection is very like .Net's, this has led to some implementations of the Java library to try and get around this for file handles by triggering the garbage collector if an attempt to get a file handle fails, then trying again. This helps that particular program avoid running out, but may still be starving the system of the handles in the meantime.

```
  public static int main() {
    B b;
    using(b = new B(1)) {
      // use b;
    } // b.Dispose called automatically
  }
}
```

Unfortunately `using` only works for objects whose lifetime is a local scope, but not for members, and they have to be cleaned up by hand.

C# doesn't allow objects embedded in other objects, only simple types and references to objects on the heap, so `c` has to be created on the heap, and this makes writing the constructor to cope with an exception being thrown more difficult.

`Dispose` has to be written by hand every time and if you forget to dispose of something, or it didn't used to be disposable but now ought to be, the resources haven't been disposed of at the right time.

An Improvement? C++/CLI

Microsoft is about to release their new attempt at getting C++ to work with CLI (the common language part of .Net). Its previous Managed C++ suffered from many problems, and is not widely used.

In this language, the solution can use many familiar C++ idioms:

```
ref class B : public A {
public:
  B(int b_param)
     : A(1), c(3), d(gcnew D(4)) {}

private:
   int b;
   C c;
   auto_ptr<D> d;
             // write one for CLI references
};

int main() {
  B b(1);
  // use b
}
```

The destructors here are `Dispose()`, and the compiler is generating the implementation and the calls, just like C++.

I've assumed that there is an `auto_ptr` analogue that works with CLI references and the rest is just the slightly different syntax for creating an object on the managed heap.

Back To The Problem

In the original system, storage for financial instruments was managed by a simple Portfolio object, which had a `Close` method to tidy it up. An instance of this was shared between several Processor objects used to manipulate the portfolio, instances of which were in turn shared between several User Interface components.

The obvious first step was to make the Portfolio implement `Dispose`, and have that close the storage.

But it was not obvious who should be disposing of this object or when – there was no clear ownership and no notion of how long the object would remain usable for – one Processor could dispose of the Portfolio and the others could then try to use it again. My solution was to push the issue of ownership and destruction up a level, by making all the Processors that used the Portfolio themselves disposable, and documented that they could use the Portfolio given to them until they themselves were disposed of.

The User Interface objects were already disposable, so it was a simple matter to pass in the Processor they needed, and again define that they could use it throughout their own lifetime.

The top-level form created the Portfolio and Processors, hooked them up to the User Interface and set everything going. Finally, in response to the form needing to close, it was then a simple matter to dispose of all the User Interface objects, dispose of the Processors, and then dispose of the Portfolio.

So here we have an interesting consequence: if a resource must be cleaned up promptly, then every object that uses it needs to think about when it is no longer allowed to use it. In this case I did it by imposing a lifetime on the Processors and User Interface objects and guaranteeing that the Portfolio would outlive them.

The consequence of having a lifetime managed by calling `Dispose` has just spread from a low level helper tucked away in some other objects, all the way up to a top level object. It is very pervasive.

In this case, the solution resulted in virtually all non-trivial classes needing to implement `Dispose`, and involved a non-trivial amount of design rework to make the ownership relations and lifetime issues clear. The only classes that were not affected were very simple "value" types used to group together data items. The language and compiler provided no help as I had to write all the `Dispose` methods by hand, call `Dispose` for every non-trivial member, and hope that if a new member is added in the future or a class becomes disposable, then the writer remembers to update the `Dispose` method.

Conclusions

Far from Garbage Collection relieving the programmer of having to think about ownership and lifetime, these issues still exist in just the same way as in C++. Only relatively simple types have no need of the `Dispose` idiom and can be left to the collector – any type that uses, directly or indirectly, resources that need to be released in a timely fashion, needs to have their relative lifetimes thought about.

Current languages such as C# don't help the programmer in writing the mechanics of these things, but the forthcoming C++/CLI will bring many of the tools that C++ provides to improving this area.

Ric Parkin
ric.parkin@ntlworld.com

References

[1] Alan Griffiths, "Heretical Java #1: Immortality – at a price", *Overload 59*
[2] http://www.artima.com/insidejvm/ed2/gc.html
[3] http://www.iecc.com/gclist/GC-faq.html
[4] http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemidisposableclassdisposetopic.asp

C++ Lookup Mysteries

by Sven Rosvall

One day, my friend Tommy asked me why his C++ code failed. He wanted to print out a number of objects (of his own class) to a stream. It worked well with a plain `for`-loop and an output operator (<<), so he knew that his output operator for the class worked as intended. But when he used `std::copy()` and `std::ostream_iterator` it failed. He wanted to "go STL" because everyone, myself included, was telling him how great the STL is.

It took us a while to figure out what was wrong and it brought us down the dark sides of the inner workings of C++. It was an interesting experience and one that I would like to share.

This article investigates function lookup in C++ and also contains a suggestion of what to do when you want to use several different output formats and still use output operators and STL.

The Code

Tommy used a class developed for a toolbox. This toolbox was declared inside a namespace, following project guidelines to avoid name collisions. Namespaces were considered good and were used a lot throughout the project.

Tommy followed the guidelines and put the toolbox client code in a different namespace. In here he had a need to print out objects of this new class stored in a container. He wrote a function that iterates over the container and an output operator for this purpose. His code was something like this:

```
namespace Client {
  std::ostream & operator<<(std::ostream &os,
                   Tools::Spanner const & s) {
    os << "Spanner{ID=" << s.getID()
       << ", gapSize=" << s.getGapSize()
       << "}";
    return os;
  }

  void printSpanners(std::ostream & os,
                  Tools::Toolbox const & tb) {
    for(Tools::SpannerCollection
             ::const_iterator sit
                   = tb.getSpanners().begin();
        sit != tb.getSpanners().end();
        ++sit) {
      os << *sit << "\n";
    }
  }
}
```

This code worked nicely. He then introduced some STL-isms and rewrote the printing function to use `std::copy()` and `std::ostream_iterator`. These functions are often together in C++ books to show the power and flexibility of the STL. An `std::ostream_iterator` is an output iterator and is used with algorithms in the same way as any other output iterator. When an object is assigned to a dereferenced `std::ostream_iterator`, this object is written to the output stream that the `std::ostream_iterator` was constructed with, using an output operator defined for that object. The `std::ostream_iterator` is specialised with a type of the objects it shall print out. The constructor of `std::ostream_iterator` can also take an optional second parameter that will be used as separator string between the printed objects. Every time an object is assigned through an `std::ostream_iterator`, that object is printed to the `std::ostream` object using the output operator.

The rewritten output operator code looked something like this:

```
void printSpanners(std::ostream & os,
                   Tools::Toolbox const & tb) {
  std::copy(tb.getSpanners().begin(),
            tb.getSpanners().end(),
            std::ostream_iterator<
                   Tools::Spanner>(os, "\n"));
}
```

Nice simple code, except that it didn't compile. The compiler could not find an appropriate output operator. The error message from the compiler was not very helpful. It said it could not find the output operator, but did not provide many clues to what it was looking for or why it could not find the output operator that is shown above.

Tommy was very puzzled, he knew that an output operator existed. He had used it successfully just a minute ago. He tried to move the output operator to the global namespace to make sure that it would be visible, but this did not work either.

When neither Tommy nor his colleagues could figure this out, he lost his enthusiasm for the STL. When we met again, he was very quick to vent his frustration with the STL in front of everyone around. I was puzzled too when I heard this story and of course I tried to defend C++ and STL. But was the problem with the compiler, the C++ standard or was there something in his code?

I asked him to come up with a small example, but he said there was too much code involved and too little time to reduce the code bit by bit while preserving the symptoms. Instead we had a discussion on how the code looked and we came up with the example above. We ran it through a couple of compilers and came up with similar error messages for all of them. So we probably could not blame the compiler. But what was wrong?

The C++ Lookup Rules

Now that I had a code example, I could play with it a bit more and read the standard thoroughly.

During lookup, operators are treated as any function, they just have a special name. The rules for finding unqualified functions and operators have two main parts. Firstly, the nearest enclosing namespace is searched for 'entities' with the same name. Note that as soon as a name is found the search stops. A function in an enclosing namespaces will be ignored even if the name found cannot be called with the arguments or if in fact it is not even a function, thus:

```
namespace A {
  void f(int);
  void g(int);

  namespace B {
    void f(double);      // hides A::f(int)
    void g(const char*); // hides A::g(int)
```

```
    void caller() {
      f(1); // calls A::B::f(double)
      g(1); // error: cannot convert '1'
            // to a 'const char*'
    }
  }
}
```

In this example we see that `A::B::f(double)` hides `A::f(int)` and is thus the only function considered in the first call. The `int` argument can be converted to `double` so this call is legal.

In the same way, `A::B::g(const char*)` hides `A::g(int)`. But the `int` argument in the second call cannot be converted to a pointer and the call is illegal.

Note that `A::g(int)` is not considered at all, even though `A::B::g(const char*)` cannot be used in the call.

After searching the current and enclosing namespaces, any functions with the same name are searched in namespaces associated with the types of the arguments to the function. This second part is called argument-dependent-lookup (a.k.a. ADL or Koenig-lookup). Consider:

```
class X {};
void f(const X &);

namespace A {
  class Y : public X {};
  void f(const Y &);
}

void caller() {
  A::Y y;
  f(y);  // calling A::f(const Y &);
}
```

Both functions `f(const X &)` and `A::f(const Y &)` are found by the lookup rules and considered for overload resolution. `f(const X &)` is found by looking at the nearest namespace and `A::f(const Y &)` is found using argument-dependent-lookup.

The argument `y` has a type defined in namespace `A` where the function `A::f(const Y &)` is found. The overload resolution rule looks at both functions and chooses `A::f(const Y &)` as a better match.

So, in the function `printSpanners()`, using the `for` loop, we find the output operator in the same namespace (`Client`). If the output operator was declared in the global namespace instead, we would find it there, unless there were other output operators in the namespace `Client`. The namespace `Tools` would also be looked at as the argument type `Spanner` is declared there, but there are no output operators there.

The problem for Tommy is that when `std::copy()` is used, the first stage of the search starts in the namespace `std`, and not in namespace `Client`. This is because the call to the output operator is from within the function body of `std::copy()`. Namespace `std` has a number of output operators as defined in the C++ standard in order to facilitate formatted output of any built-in type and some types defined in the C++ library. It doesn't matter that none of these overloaded output operators can be used with `Spanner`. The lookup rule says that we find the function in the nearest enclosing namespace and stop. The output operator defined in the namespace `Client` is not considered at all as this namespace is not an enclosing namespace of namespace `std`. The compiler won't even find the output operator if it was defined in the global namespace as it had already found some output operators in namespace `std`, its nearest namespace.

Had Tommy declared the output operator in the same namespace as the class (namespace `Tools`), he would have avoided this problem as the second rule (ADL) would have found it. It can be seen as part of the interface of the class and should be declared close to the class itself, preferably in the same header file. This is fine if you have control over the header file. It does not work if the header is part of a third party library. As a workaround it is possible to put the declaration in any header file by re-opening the namespace like this:

```
namespace Tools {
  std::ostream & operator<<(std::ostream & os,
                            Spanner const & s) {
    ...
  }
}
```

The Real Problem

So what was Tommy trying to do? Why was the output operator declared in the `Client` namespace and not in the `Tools` namespace where it belongs? Tommy said that he could have added the output operator in the `Tools` namespace, but he wanted different output formats for different client applications. He couldn't place the output operators beside the `Spanner` class definition as you can only have one of them in the same namespace. There is no way to overload two output operators with another parameter. For his project it was easy to use namespaces to separate the output operators as no client in the same namespace would use more than one format.

A Solution to the Real Problem

So how can we make a design where we can have different output formats? How can we use these formats using output operators? And how can we make a design that will work when we use `std::copy()` and `std::ostream_iterator`?

To start this off, we want some way to select different formats when a `Spanner` object is printed to a stream. Possibly you could derive from `Spanner` and then overload the output operator on these derived classes. Not a very nice design and it won't work as you cannot downcast a `Spanner` object to the derived class.

A simpler approach is to use different named functions that do the formatting. We want a simple syntax such as:

```
std::cout << spanner.printNameAndGap()
          << std::endl;
```

This can be implemented by letting the member function `printNameAndGap()` return a string in the format we want.

Nice and simple. Except that it is not always possible to add things to the class we are using, for example third party libraries. Here, the formatting belongs to the user, not to the class itself. The class designer does not know what format all clients can possibly want to use. This approach is also inefficient, as a temporary string has to be created.

Instead we want to use a non-member function and we want writes made directly to the output stream. This function can return an object of a class that can be used with an overloaded output operator. To make it easy, we use the constructor of this formatting class instead of a separate function.

```
class PrintSpannerNameAndGap {
public:
  PrintSpannerNameAndGap(Spanner const & s)
      : m_s(s) {}
  void print(std::ostream & os) const {
    os << "Spanner{ID=" << s.getID()
       << ", gapSize="
       << s.getGapSize()
       << "}";
    return os;
  }
private:
  Spanner const & m_s;
};

std::ostream & operator<<(std::ostream & os,
                PrintSpannerNameAndGap
                          const & spanner) {
  spanner.print(os);
  return os;
}
```

We can now use this class like this:

```
std::cout << PrintSpannerNameAndGap(spanner)
          << std::endl;
```

This does not look too bad. Just watch out for that member reference to the original object. The `PrintSpannerNameAndGap` object must not exist longer than the referenced `Spanner` object. This is not a problem when it is used as shown above as it only exists as a temporary object and disappears at the end of the statement.

Using `std::copy()` and `std::ostream_iterator`

`std::copy()` is nice but it is not possible to insert a formatting object in the way shown above. We have to look at other ways to indicate that we want different output.

If we look at the line using `std::copy()` and `std::ostream_iterator` there aren't many opportunities for modification. We could adapt the source iterators (the `begin`/`end` pair) to return a different object when dereferenced, and define an output operator for each different object type. The mechanism for choosing the correct overloaded output operator would be similar to the approach above.

But there is no need to create the iterator adaptor. We only have to specify to the `std::ostream_iterator` that it shall work with `PrintSpannerNameAndGap` objects. This makes the code much simpler:

```
std::copy(tb.getSpanners().begin(),
          tb.getSpanners().end(),
          std::ostream_iterator<
                 PrintSpannerNameAndGap>(
                              os, "\n"));
```

`PrintSpannerNameAndGap` is the same class as above. As the `std::ostream_iterator` requires a `PrintSpannerNameAndGap`, then the `Spanner` objects returned by the source iterators are implicitly converted to `PrintSpannerNameAndGap` objects. This works because we did not make the constructor `explicit`. The `PrintSpannerNameAndGap` object is a temporary object and is deleted after the assignment statement in `std::copy()` has completed. The `PrintSpannerNameAndGap` object holds a reference to the `Spanner` object coming from the iterators to avoid unnecessary copying. This reference is OK as the `PrintSpannerNameAndGap` object has shorter lifetime than the `Spanner` object.

Possible Improvements

We could use templates to reduce the amount of boilerplate code. But introducing templates does not reduce enough code to motivate the extra complexity.

Another approach would be to let the formatting class `PrintSpannerNameAndGap` inherit from a base class that is used by all classes supporting different output formats. This base class would keep the reference to `Spanner` and declare the function `print()` pure virtual. A single output operator definition for this base class replaces all specific output operators. This only pays off when there are many different output formats for the same object type.

A specific functor object can be used with the C++ library algorithm `std::for_each()` to print out each element. Initialise the functor with the output stream and define an `operator()(Spanner const &)` that prints each object to the output stream in the required format.

Conclusion

It is not always easy to understand what happens under the hood in C++. But there are solutions to every problem even if good understanding of C++ may be required. Don't be afraid of asking friends or other ACCU members for advice.

Sven Rosvall
sven-e@lysator.liu.se

Acknowledgements

Thanks to Tommy Persson who had the problem originally and spent time describing the problem to me, to Richard Corden for clarifying the C++ standard and to Thaddaeus Frogley for reviewing.

[The name lookup rules in C++ are not only confusing for the non-expert, they present serious problems to the expert. One such expert (Dave Abrahams) has presented these problems, and proposed a solution, for consideration by the standards committee:
`http://www.boost-consulting.com/writing/n1691.html`
– Alan (ed)]