contents

Copy Deadlines

All articles intended for publication in *Overload 73* should be submitted to the editor by May 1st 2006, and for *Overload 74* by July 1st 2006.

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@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Tim Penhey
tim@penhey.net

Advertising:
Thaddaeus Frogley
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: Doing What You Can

Your magazine needs you!

If you look at the contents of this issue of Overload you'll see that most of the feature content has been written by the editorial team. You might even notice that the remaining article is not new material. To an extent this is a predictable consequence of the time of year: many of the potential contributors are busy preparing for the conference.

However, as editor for the last couple of years I've noticed that there is a worrying decline in proposals for articles from authors. Many of the great articles you've enjoyed over the last year have only arisen because I've happened to be talking to someone about a problem they have encountered developing software and the way they solved it. When I suggest that they write it up the usual response is "no-one would find that interesting" – but those that have gone ahead and written an article will attest that people are interested in these things. We belong to a community that enjoys solving the problems of software development and in hearing about the experience of others.

One of the things about a community is that it is strengthened by sharing, all of you have stories to tell – and the purpose of Overload is here to help you tell them. The team of advisors listed at the front of Overload are not here to write the magazine for you, they are here to help with the selection of the best stories to publish and to work with the authors to ensure that their stories are told well. And after the advisors' efforts to fill the pages of Overload this time I'm going to enforce a rest. I will not be accepting any articles from the advisors for the next issue of Overload. (If the next issue consists only of an editorial then you'll know why.)

I know, because I've heard the stories, that several of you could write about "Proving Brookes' Law" (the industry clearly needs more evidence); or "A Pair Programming experience report" (that may be better written jointly– when you are talking to each other once more); or "Doing the Right Thing" (when judgment and policy disagree). I'm sure that others I've not talked to could also contribute to these or other stories.

Make an effort to contribute to the magazine. If you are willing but "don't have anything to write" then seek me (or one of the advisers) out at the ACCU conference, at XtC, at SomethingInNottingham, or by email – every one of you has a story to share, you just might not realise it yet. It doesn't have to be a success story: "cautionary tales" are as informative as "happy endings" (and sometimes more useful).

Remember, I'm not going to let the advisors write Overload 73 for you. It is your magazine and you need to make an effort.

More on WG14

Last time I commented on the fact that I was hearing conflicting information about the WG14 technical report on "safe" alternatives to the standard library function. This has led to an email from the Convener of WG14 (John Benito <jb@benito.com>):

> Reading the editorial in the February 2006 edition of the Overload magazine written by you, Exhibit 3. WG14 is indeed working on a Technical Report that will define more resilient library functions (we are staying away from "secure", "safe" and that ilk). This work was first brought to WG14 by Microsoft, but to indicate it would have Microsoft specific extensions embedded is just ridiculous and a statement like that would only come from the misinformed. Chris Hills indeed was the Chair of the UK C Panel, but attended (or partially attended) just one (yes one) WG14 meeting his entire tenure as UK C Panel convener, and should not be used as a reference to what is happening within the Working Group. If you want to know what is going on in WG14, ask me, or Francis Glassborow, or at the very least ask someone that actually attends the meetings.
>
> I would also like to point out that using statements like "later this year if WG14 (article has WG13) accepts this" makes it seems as if you do not understand the ISO process. WG14 generated an NP for this work in 2003, this NP was balloted at the SC22 level, passed and WG14 officially started work on the project. Since then the TR has been through the Registration ballot phase and is ready for the next ballot in the process. Again, if you are not sure of this process ask. Also implying anyone can write a Technical Report because we are all volunteers is misleading and just not the case. There is a process, and the process is followed.
>
> I am not trying to harsh in any way, but I really do not want WG14 misrepresented to the Community.
>
> Thank you. John Benito (WG14 Convener)

I hope that clarifies the situation. I apologise for implying that due process is not being followed and only intended to observe that, typically, national bodies do not actively oppose work that they themselves don't wish to spend time on.

More on C++/CLI

I've also had an email about C++/CLI. Mark Bartosik <mbartosik@yahoo.com> writes:

> Alan, I've just read your editorial in Overload. A lot of what you said rang very true with me.
>
> I've got a reasonable degree of competence in C++ (attended ACCU conference for about ten years, and speaking this year). I say

this to place into context my following comments...

When I first heard from Herb a couple of years ago what Microsoft were doing to bind C++ to CLI it sounded cool. However, when I now look at what it has evolved into I too feel like it is a different language, for me it is the attributes and keywords where library code might have done the job.

Now for the important comment that is also an embarrassment....

When I read some of the Visual Studio 2005 / MSDN documentation, I did think that I had lost track of what was happening to the C++ standard. I thought that some of what were in reality C++/CLI features were C++ features that were in TR2 and I had somehow got confused about what was in TR2. No it was just that the documentation is highly misleading, and it mislead me. Not for long, but I'm sure that many of my coworkers would have been completely fooled.

I have no experience in C++/CLI, indeed I've only read some of the documentation, but I was fooled by the documentation for at least a short time. The documentation does make it feel like a different language but also gives the impression that some features are C++, some are plausible features (attributes are obviously not).

Nice editorial, Mark

The issue that should concern us as developers is the confusion that Mark highlights regarding which semantics belong to C++ and which to C++/CLI. However, ECMA's response to the issues raised by BSI dismisses this:

> There is no reason to expect that programmers will mistake C++/CLI features for ISO C++ features any more than they occasionally mistake extensions in specific compilers (e.g., Borland, Gnu, and Microsoft) for ISO C++ features. Rather, having these extensions as a named standard instead of unnamed vendor extensions serves to help distinguish them more clearly from ISO C++.

This confusion of language semantics would not arise with a library based "binding" to the CLR – or with language extensions to add semantics. The difficulty of doing this is discussed in Herb Sutter's excellent paper "A Design Rationale for C++/CLI": to achieve the design goals that the C++/CLI group set themselves language support for a radically different programming model is necessary.

Those involved in the "C++/CLI Language Specification" have done a good job of solving the problems set by integrating C++ into a platform with a radically different object model: C++/CLI is better thought out than comparable technologies like Microsoft's "managed C++" or Borland's VCL. It does look like a good and useful addition to the curly-brace family of languages! (And C++/CLI is definitely "on topic" for Overload – so don't be shy of writing articles that use it.)

In any case, it seems likely that around the time you are reading this, there will be more developments on this front. Both BSI representatives and Microsoft representatives will be present at a WG21 meeting in Berlin and we can expect that a constructive dialogue will ensue.

More on the Safe Standard C++ Library

A great deal of effort has been expended explaining to the Microsoft representatives that others may have a different security model than they do (and, therefore, different vulnerabilities), that other concerns dominate in other organisations (which make the suggested "solution" unacceptable), that the diagnostics may be issued for manifestly correct code, that changing code to fix the diagnostic can result in incorrect code, and that the choice of wording creates a misleading impression regarding the authority of the diagnostics.

Obviously, there are limitations to what Microsoft can do with software that has already shipped, and even to what they can change in a service pack. But clearly things have been happening within Microsoft, because Herb Sutter (yes, he's been busy) recently came back to the WG21 reflector seeking feedback on their current plans for addressing this in a service pack. In particular, the wording has been changed to avoid implying that the advice is standards based, and to make it easier for library vendors to suppress the diagnostics appropriately. While there are concerns that have not been addressed by this proposal it is a big improvement.

Probably the most significant concern is due to a lot of the diagnostics produced being false positives. It is still easy to envisage scenarios where correct code is flagged as being "unsafe" resulting in a developer spending time to fix the "problem" and, in the process, producing "safe" code that is incorrect. While I accept that scenarios like this should be rare, they will be noticed: "you broke it because the compiler said it was wrong? That doesn't happen with <insert random programming language here>". Once a few stories like this start "going around" the facts will become irrelevant – like the idea that C++ programs have memory management problems and that Java programs don't.

In Conclusion

Each of us can only do so much to uphold the values we espouse. But we have a solemn duty not only to do what we think is right, but to recognise what others are achieving. Sometimes the latter may not be obvious but, given the history of Microsoft, does anyone believe that there would be any progress towards resolving the above issues without considerable internal resistance?

I hope to see you at the conference!

Alan Griffiths
overload@accu.org

References

1. ACCU conference: http://www.accu.org/conference/
2. XtC: http://www.xpdeveloper.net/xpdwiki/Wiki.jsp?page=XtC
3. SomethingInNottingham: http://www.xpdeveloper.net/xpdwiki/Edit.jsp?page=SomethingInNottingham
4. ECMA's response: http://www.octopull.demon.co.uk/editorial/ECMA_comments_on_ISO_IEC_DIS_26926_C++_CLI_Language_specification001.pdf
5. "A Design Rationale for C++/CLI", Herb Sutter http://www.gotw.ca/publications/C++CLIRationale.pdf
6. "C++/CLI Lanaguage Specification" - http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf

Multithreading 101

by Tim Penhey

Multithreading is a huge topic and this article is really just giving an introduction to many terms and concepts. It is aimed at the threading newcomer and intends to highlight some of the problems associated with writing multithreaded applications.

Firstly I'll cover the basics on threads – what are they, and how do they relate to processes. Next there are some guidelines of where threads can be useful, and when not to use them. Then I'll explain the "primitives", the underlying building blocks provided by the operating system kernel to allow multithreading programs to be written.

Following that I cover some concepts that do not at first seem intuitive: memory barriers – what they are and why they are needed; race conditions – with specific examples of deadlock and livelock; and priority inversion – where a low priority task gets more processor time than a high priority task.
Lastly a look at the where the world of threading is headed.

Tasks, Processes and Threads

In the beginning there was synchronous execution. A list of commands were executed in order, one at a time. As time progressed, more and more was expected from an operating system. Multitasking allowed multiple programs to appear to be running at the same time, while in actual fact the CPU was still only capable of running one thing at a time.

In order to switch computation from one task to another there would be a context switch. The execution details of one task were saved out of the registers in the CPU, and details of another task were moved in, and execution of the new task would resume. Multitasking appeared in two forms: cooperative and pre-emptive. With cooperative multitasking, it is up to the application to relinquish control, whereas with pre-emptive multitasking the OS kernel will interrupt the task "mid flow" and switch it out. The primary problem with cooperative multitasking is that it only takes one badly behaved process to disrupt the entire system.

Processes can be thought of as tasks that have certain isolation guarantees with respect to memory and resources. Threads are created by a process – from its "main thread". Created threads share memory and resources with the thread that created it.

Shared access to memory and resources combined with pre-emptive multitasking is where the complexity of multithreaded programming comes in.

When to Use Threads

Unfortunately there is no black and white answer for this question. There are several important questions that you should ask yourself before diving into multiple threads:

- Are shared resources really needed?
- Is the extra complexity worth the effort?

There are many cases where it seems desirable to have separate threads – particularly when there may be clean separation in processing. An important question here is "should they be separate threads or separate processes?" Do both tasks really need access to shared resources or is there just information being passed from one to the other? Two single threaded programs are often better than one multithreaded program due to the following points:

- Single threaded code is easier to write – no resource synchronization needed
- Single threaded programs are easier to debug
- Single threaded programs are easier to maintain – no threading knowledge needed
- Separate programs are easier to test – each can be tested in isolation

Threading adds complexity to often already complex situations. A good rule of thumb is "Is the complexity that is added through the use of threads significantly reducing the complexity of the problem?" If the answer is "NO" then you may want to strongly reconsider the use of threads. However there may be other overriding factors that push strongly towards a threaded solution:

- GUI responsiveness
- Complete processor utilisation
- Network connectivity

GUI Responsiveness

By their nature GUI programs are event driven. For much of the time, the programs are waiting for the user to click or type something. GUI programs are normally written with a main event loop where the application needs to respond to "desktop events" such as repaint or move as well as user events. When applications fail to respond to these events in a timely manner, the operating system often marks the processes as non-responsive and the application can appear frozen.

There are often situations in writing an application where long running functions are needed (for some arbitrary definition of long) – for example: reading or writing files, running a spell check, or encoding some data. In order to have a responsive GUI for the application it is not desirable to have these functions running in the same thread as the main event processing loop.

Complete Processor Utilisation

Some classes of problems lend themselves to being easily broken into independent parts. Some classes of problems need to be solved as soon as possible. Some classes of problems are purely calculation intensive where the problem is not waiting on other results or information from other sources. The intersection of these three sets of problems are candidates for a solution where all the processors in a machine could be fully utilised. In situations where separate processes are not viable, having a number of threads to do the processing will provide a time benefit. One of the tricks is not to have more processing threads than effective CPUs (effective as a single hyperthreading CPU can appear as two). The reason for this is that if all the threads are CPU bound (which means the only thing slowing the calculation of the results is the speed of the processor), then ideally there should be one thread assigned to each effective CPU. If there are more calculation threads than CPUs then more context switching occurs as the active threads compete with each other for the processors. Every context switch has an overhead which slows down the results generation even more.

Network Connectivity

Threads used in this way are often referred to as "worker threads". A point to consider when dealing with worker threads is when to create them. There are always overheads involved when creating and destroying threads. The amount of overhead varies from platform to platform. It is a common idiom to use a collection of threads that are started prior to any work being available for them. These threads then wait for work to do. The collection of threads is often referred to as a "thread pool" or "work crew". Since the thread generating the work does not pass it on to a specific thread, there needs to be some abstraction between the threads generating the work and the threads processing the work. This is often implemented using a queue. The work generating threads put work on the queue, and the workers take work from the queue. In situations such as this there are normally two types of work: those that have feedback – results passed back to the work generators; and those that do not – for example the worker threads save results directly to a database. How the feedback from the worker threads is handled depends greatly on the problem – suffice to say there are many ways to do it.

Whether on the client or the server side, threads are often used when communicating over a network. On the client side threads are used when requesting a response that may take the server some time to respond to – especially if the client side application has more to do than just wait for the response such as responding to GUI events.

Threads are more common on the server side where the server may be listening on a socket. When a client connects a socket descriptor is generated that represents the connection between the client and the server. The socket descriptor can then be used independently of waiting for more connections. In order for the server application to remain responsive to client connections, the actual work with the socket descriptors is normally handled by a different thread than the thread where the program is accepting connections.

Thread pools are frequently used to handle server side execution where the threads wait for the primary thread to accept a client connection. The socket descriptor is then passed to an idle worker thread in the thread pool. This thread then handles the conversation with the client and performs the necessary actions to produce results for the client. Once the conversation with the client is finished, the worker thread releases the socket descriptor and goes back into a waiting state.

Primitives

Providing an API to start, stop and wait for threads is really just the tip of a much bigger iceberg of necessary functions. The simplest of these functions are atomic operations. These are functions, provided by the OS kernel, that are guaranteed to complete before the task is switched out.

As well as atomic additions and subtractions, there are also functions that perform more than one "action" such as "decrement and test" and "compare and swap".

- Decrement and test subtracts one from the atomic variable and returns true if the atomic variable is now zero.
- Compare and swap takes three parameters, the atomic variable, the expected current value, and the new value. If the atomic variable is the expected value then it is updated to the new value and returns true. If the atomic variable is not the expected value, then the expected current value is updated to what the atomic variable currently holds and the function returns false.

These atomic operations are then used to build more complex primitives.

In order to provide some isolation guarantees for multiple threads within one process, the operating system provides a way to define mutual exclusion blocks. The primitive that provides the interface to these blocks is often referred to as a **mutex** (*mut*-ual *ex*-clusion). The key to how the mutex works is in the acquiring and releasing. When a mutex is acquired, it is effectively marked as being owned by the thread that acquired it. If another thread attempts to acquire the mutex while it is still owned by a different thread, the acquiring thread blocks waiting for the mutex to be released. When the mutex is released it is marked as "unowned". In the case where there is another thread waiting, it is effectively woken up and given ownership of the mutex. Some care needs to be taken acquiring mutex ownership as some implementations will not check for self ownership before waiting, so it is possible for a single thread to wait for itself to relinquish ownership – not a situation you really want to be in.

Any non-trivial data structure uses multiple member variables to define the structure's state. When the data structure is being modified it is possible that the thread that is updating the data structure could be switched out. Another thread that wants to access the shared data structure could be switched in. Without the use of mutual exclusion blocks it is possible that the data structure's state is inconsistent. Once this happens the program is often in the world of undefined behaviour.

Consider the following example:

```
There is a container of Foo objects called
foo.
A mutex to protect foo access called m

Thread 1:
   acquire mutex m
   bar = reference to a value in foo
   release mutex m
   use bar...

Thread 2:
   acquire mutex m
   modify foo
   release mutex m
```

What is protecting **`bar`**? It is possible that thread 2 may modify the value of **`bar`** while thread 1 is trying to use it. Protecting memory access between threads is almost always more complex than it first looks. One approach is to put mutexes around everything, however this does little more than serialise the application and prohibits the concurrency that is intended through the use of threads. Multiple threads with extraneous mutexes often perform worse than a single threaded approach due to the additional overhead of context switches.

The next two primitives are more for synchronization than mutual exclusion. They are **`events`** and **`semaphores`**. Events and semaphores can be used to initiate actions on threads in response to things happening on a different thread of execution.

Unfortunately there is no agreed standard on what an event is, and the two most common platforms (Win32 and posix) handle things quite differently. Without going into too much detail, you can have threads waiting on an event. These threads block until the event occurs. When the event is signalled (from a non-blocking thread obviously) one or more threads (implementation dependant) will wake up and continue executing.

An example of using an event would be a client-server system where the client is able to cancel long running calls. One way of doing this is to have the thread that is communicating with the client not actually do the work, but instead get another thread to do the work and be told when the work is complete. The way in which the servicing thread is informed of complete work is through the use of an event.

```
Thread 1:
   gets client request
   passes on to worker thread
   waits on event

Worker thread:
   gets work for client
   processes for a long time
   signals event when complete

Thread 1:
   wakes on event signal
   returns result to client
```

What advantage does this give over just one thread? Implemented this way allows the possibility of having either time-outs for long work or client driven cancellation. For client driven cancellation it may work something like this:

```
Thread 1:
   gets client requesting
   passes on to worker thread
   waits on event

Worker thread:
   gets work for client
   processes for a long time

Thread 2:
   client cancels original request
   sets cancelled flag
   signals event

Thread 1:
   wakes on event signal
   returns cancelled request
```

What happens to the worker thread depends on the problem. The worker thread may be allowed to run to completion and discard results or it maybe be possible to interrupt the worker thread prior to completion.

For time-outs instead of having a client connection initiating the cancellation it is some "watchdog" thread whose job it is to observe running work and cancel work that runs overtime.

A semaphore can be thought of as a counter that can be waited on. A semaphore is initialised with a non-negative number. When a thread wants to acquire a semaphore, this value is checked. If the value is positive, then the value is decremented and execution continues. If the value is zero, then the thread blocks. The semaphore value is incremented by a non-blocked thread calling the "signal" or "release" semaphore function. If there is a waiting thread, it will awaken, decrement the value and continue executing.

Semaphores are often used where there are multiple copies of a resource to be managed. For example a database connection pool. Instead of initiating a connection every time the database needs to be accessed, a number of connections are created "ahead of time" and await use. One way of managing this pool of connections is to create a semaphore with an initial count of the number of available connections. When a thread requests a connection it does so by acquiring the semaphore. If the value of the semaphore is non-zero the semaphore is decremented which means there is a database connection available. When the thread is finished with the connection, the connection is returned to the pool and the semaphore is incremented. If the semaphore is zero when a connection is requested, the thread blocks until the semaphore is incremented. When a connection is returned to the pool and the semaphore incremented a thread waiting on the semaphore will awaken – decrementing the semaphore in the process, allowing it to get the connection.

Memory Barriers

Advances in processor efficiency have led to many tricks and optimisations that the processor can do in order to speed up memory access. Grouping together memory reads and memory writes, along with block reads and writes are some of the tricks that are done to make access faster. The reordering of memory reads and writes are a special case of the more general optimisation referred to as "execution out of order". The commands are grouped in such a way that any grouping that is done is transparent to a single thread of execution. The problem occurs when there is more than one thread sharing the memory that is being read or written to in the optimised manner.

Suppose we have two threads as illustrated by this pseudo code:

```
x = 0, y = 0

Thread 1:
   while x == 0 loop
   print y

Thread 2:
   y = 42
   x = 1
```

It might seem reasonable to expect the print statement from Thread 1 to show the value 42, however it is possible that the processor may store the value for x before y, and hence the printed value may show 0.

A memory barrier is a general term that is used to refer to processor instructions that inhibit the "optimal" reordering of memory access. A memory barrier is used to make sure that all memory loads and saves before the barrier happen before any loads

and saves after the barrier. The actual behaviour of memory barriers varies from platform to platform.

Many synchronization primitives have implicit memory barriers associated with them. For example it is usual for a mutex to have a memory barrier defined at the point of acquiring and at the point of releasing.

As well as processor execution out of order, there are also issues with compiler optimisations. What a compiler may do depends on the memory model for the language that is being used. The more thread aware languages provide keywords which will stop the compiler making certain assumptions, for example the volatile keyword in Java.

Race Conditions

A race condition is a catch-all term for situations where the timing of the code execution can impact the result. The definition provided by the Free On-Line Dictionary Of Computing [1] is "Anomalous behavior due to unexpected critical dependence on the relative timing of events".

Many of the well known problems associated with multithreaded code are types of race conditions. A large number of race conditions can be solved through the use of synchronization primitives. However overly liberal use of synchronization primitives will have a performance impact and can also bring a different type of problem into the picture.

Probably the best known result of a race condition is **deadlock**. A deadlock is where two or more threads are waiting for resources that the other waiting threads own – so each waiting thread waits forever.

There are four necessary conditions for deadlocks to occur. These were initially defined in a 1971 paper by Coffman, Elphick, and Shoshani [2].

1. Tasks claim exclusive control of the resources they require ("mutual exclusion" condition).
2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no pre-emption" condition).
4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

Having these conditions in the code does not necessarily mean that deadlocks will occur, just that they could, and hence the relative timing of events in different threads can lead to this race condition happening.

There are several strategies for avoiding deadlocks, the simplest is to avoid the circular chain. This is normally achieved by specifying a particular order in which locks must be acquired if multiple locks are needed.

A less well known problem is that of **livelock**. A livelock is where two or more threads continuously execute but make no progress. For example two threads are both doing calculations while some value is true, but they are doing it in such a way that each invalidates the other's test, causing both threads to continue for ever.

A real world example of live lock is two people walking towards each other in a corridor. Each person trying to be polite steps to the side, however they just end up mirroring the other person's movement and end up moving from side to side. Luckily we don't normally end up stuck forever in this predicament.

Priority Inversion

Priority inversion is more of a problem for real-time systems, but it can occur on any system where threads or processes can have priorities assigned to them. This ends up being a scheduling problem and occurs when a low priority task has ownership of a resource that a high priority task needs.

In these situations the low priority task ends up taking precedence over the high priority task. This can be complicated even more by there being another task running with a priority between the high and low task. The medium priority task takes precedence over the low priority task which delays even longer the releasing of the resource that the high priority task needs.

In many cases, priority inversion passes without much notice, but it can cause serious problems, as in the case with the "Mars Pathfinder" [3] where the craft was resetting the computer frequently.

Where To From Here?

We are now in the situation where multi-core processors are becoming commonplace. In order to write software that will make full use of new hardware, multithreading is going to be used more and more.

Languages that have defined memory models are considering them with respect to changes in hardware technology. Languages that don't, like C++, are looking to define them. A particular interest is the possibility of adding language constructs that will allow the automatic use of threads to speed up loops.

Another area of research that is likely to see more interest is lock-free data structures and wait-free algorithms. Lock-free data structures are possible to write by using the atomic operations supplied by the operating system. However due to the more inherent complexities of non-trivial data structures, they are hard to get right.

Wait-free algorithms are of particular interest to real time systems due to the fact that if a high priority task is never blocked, then priority inversion does not happen.

Conclusion

If you are looking at using multiple threads first stop and think. Are they really necessary? Is the added complexity from threading going to reduce the complexity of the problem? Would separate processes be better? If after careful consideration you decide that they are in fact going to help, then go forward with care. Protect your shared data structures with mutual exclusion blocks or cunningly-devised lock-free algorithms. Avoid circular dependencies on mutexes, and use the correct synchronisation methods for the problem.

Tim Penhey
tim@penhey.net

References

1 http://foldoc.org
2 http://www.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf
3 http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html

To Grin Again

by Alan Griffiths

In the latter years of the last millennium there were a number of articles published that explored the design space of "smart pointers". That was in the days where there were loads of broken implementations available for download from the internet, but working ones – like those in the Loki and Boost libraries – were not to be found.

One of these pointers has come back to haunt me – partly because I wrote it, but also because it meets a set of recurring requirements. I'd not looked at it for years, but a colleague was implementing a "Cheshire Cat" class using Boost's `shared_ptr` [1] (which is a dubious choice as the copy and assignment semantics are completely wrong for this purpose) so I pointed him to my old article on `grin_ptr` [2]. This article has also been referenced by a couple of recent Overload articles, so I thought it was time to revisit the subject.

The first thing I discovered is that I've never submitted the article for publication in Overload (instead a heavily annotated version written with Allan Newton appeared in Overload 38 [3]). The second thing I discovered is that there were a number of improvements I'd like to make to both the code and to the article. What follows is the result.

The "Classical" Cheshire Cat

The "Cheshire Cat" idiom is a great solution to a set of problems that have existed since the pre-history of C++. This idiom first emerged in the late '80s in a cross-platform GUI class library called CommonView and was described by John Carolan [4]. It has been reinvented a number of times since and is also known as "Compilation Firewall" and as "Pimpl". The "Gang of Four" [5] classifies it as a special case of the "Bridge" pattern. These problems addressed by Cheshire Cat stem from three areas:

1. Ensuring that a header file includes nothing that the client code needn't be aware of. (In the case of CommonView, this includes the native windowing APIs.)
2. Making it possible to distribute updated object code libraries without requiring the client code to be recompiled. If a class definition (or anything else in a header) changes, then any compilation unit that includes it must be recompiled.
3. Protecting intellectual property by concealing implementation techniques used.

Which of these is most important to you will differ according to circumstance, but the "Cheshire Cat" idiom addresses them all.

When Glockenspiel (the suppliers of CommonView) started developing their cross-platform GUI library they were influenced by all three of the above reasons for hiding the implementation. Here is a simplified telephone list class implemented in the style of the CommonView library:

```
struct phone_list_implementation;

class phone_list
{
public:
  phone_list(const char *name);
  phone_list(const phone_list& rhs);
  phone_list& operator=(
   const phone_list& rhs);
  ~phone_list();

  const char* list_name();
  const char* find_number(const char *person);
  void  add(const char *name,
   const char *number);

private:
  phone_list_implementation* grin;
};
```

Note the need for a copy constructor and assignment operator. These are required because `phone_list` owns the `phone_list_implementation` to which it holds a pointer. (The default behaviour of just copying the pointer value is obviously inappropriate.)

In the implementation file we can imagine that the `phone_list` member functions are implemented like this:

```
const char* phonelist::find_number(
 const char* person)
{
  return grin->number(person);
}
```

The Motivation for grin_ptr

Once you've written a few classes using this idiom it will occur to you that you are writing the same functions again and again to manage the implementation. Specifically the copy constructor, assignment operator and destructor of a Cheshire Cat class are generic - they always look the same, they only differ in the types of the interface and implementation classes.

This sounds like a job for a template. A template that looks after an implementation object allocated on the heap, and ensures it is copied or deleted when appropriate. It is tempting to reach for the standard library, but the closest thing we find there is `auto_ptr<>`. Even moving further afield to Boost we don't find a smart pointer with the desired semantics.

Let us suppose for a moment that such a smart pointer exists and is called `arg::grin_ptr<>`. This would allow the `phone_list` class to be rewritten as follows:

```
class phone_list
{
public:
  explicit phone_list(
   std::string const& name);

  std::string name() const;

  std::pair<bool, std::string> number(
   std::string const& person) const;

  phone_list& add(std::string const& name,
   std::string const& number);

private:
  class implementation;
  arg::grin_ptr<implementation> grin;
};
```

Note that there is no longer a need to supply the copy constructor, assignment operator, or destructor as we are assuming that the necessary logic is supplied by the `grin_ptr<>` template. (I've also taken the opportunity to use a more contemporary style of C++, by nesting implementation, using `const` and `std::string` but the ideas are the same.)

In the implementation file we can imagine that the `phone_list` member functions are implemented the same way as in the original. For example:

```
std::pair<bool, std::string>
phonelist::number(
 std::string const& person) const
{
  return grin->number(person);
}
```

The idea is for `grin_ptr<>` to act "just like" a pointer in all relevant ways while taking the burden of ownership management. Naturally, "all relevant ways" doesn't include such things as support for pointer arithmetic since this is inappropriate for pointers used in implementing "Cheshire Cat".

By using a compatible substitute for a "real" pointer we avoid the need to write repetitive boilerplate code. Everything else is the same.

The resulting simplification looks good, but can such a smart pointer be implemented?

Implementing arg::grin_ptr<>

The principle problem to overcome in implementing `grin_ptr<>` is that the compiler generated copy constructor, assignment operator and destructor for `phone_list` will each instantiate the corresponding code for `grin_ptr<>` in a context where `implementation` is an incomplete type.

This means that the `grin_ptr<>` copy constructor cannot simply copy construct an instance of `implementation`. Likewise, the destructor of `grin_ptr<>` can't be a simple `delete p;` as deleting a pointer to an incomplete type gives undefined behaviour whenever the type in question has a non-trivial destructor. The assignment operator, will either have to deal with both these issues or delegate these problems. Let's write a test case for these operations:

```
struct implementation;
int implementation_instances = 0;
int implementation_copies_made = 0;

void test_incomplete_implementation(
  const arg::grin_ptr<implementation>&
   const_grin)
{
  assert(implementation_instances == 1);
  assert(implementation_copies_made == 0);
  {
    // Check that copy construction works
    // creates a new instance
    arg::grin_ptr<implementation>
     nonconst_grin(const_grin);
    assert(implementation_instances == 2);
    assert(implementation_copies_made == 1);
    // Check assignment calls copy constructor
    // (and that the discarded implementation
    // is deleted)
    nonconst_grin = const_grin;
    assert(implementation_instances == 2);
    assert(implementation_copies_made == 2);
  }
  // Check destructor cleans up instances
  assert(implementation_instances == 1);
}
```

Note that this test case is unable to initialise `grin_ptr<>` with an instance of `implementation` – this is a direct consequence of `implementation` being an incomplete type.

In implementing `grin_ptr<>` I make use of the fact that the constructor is instantiated in the implementation file for `phone_list`, where the class definition for `implementation` resides. Similarly calls to `implementation` member functions also require a complete type. These are tested in the next part of the test harness:

```
struct implementation
{
  struct const_tag {};
  struct nonconst_tag {};

  const_tag function() const
    { return const_tag(); }

  nonconst_tag function()
    { return nonconst_tag(); }

  implementation()
   { ++implementation_instances; }

  implementation(const implementation&)
    { ++implementation_instances;
      ++implementation_copies_made; }

  ~implementation()
   { --implementation_instances; }
};

void test_implementation()
{
  assert(implementation_instances == 0);
  assert(implementation_copies_made == 0);

  {
    arg::grin_ptr<implementation>
     grin(new implementation);
    assert(implementation_instances == 1);
    assert(implementation_copies_made == 0);

    test_incomplete_implementation(grin);

    // Check that copy construction works
    // creates a new instance
    const arg::grin_ptr<implementation>&
     const_grin(grin);
    assert(implementation_instances == 1);
```

```
    // if const qualification is wrong then
    // these should produce compiler error
    implementation::const_tag
     const_test = const_grin->function();
    implementation::nonconst_tag
     nonconst_test = grin->function();
  }
  // Check destructor cleans up instance
  assert(implementation_instances == 0);
}
```

If you examine the code for `grin_ptr<>` (shown in listing 1) you'll see that each instance carries around a pointer to a structure containing function pointers `do_copy` and `do_delete`. This structure is initialised using a trick I first saw used by Richard Hickey [6]: the constructor (which is instantiated with a complete type) instantiates the corresponding `my_delete_ftn` and `my_copy_ftn` template functions and stores the pointers to them. Because these see the full definition of the class used to initialise the pointer they can make use of its copy constructor and destructor (the casts are there to support implementation hierarchies). Using pointers to functions provides a safe method for `grin_ptr<>` to copy and delete the object it owns. The point of passing around function pointers instead of the apparently more natural use of virtual member functions is that everything can be done "by value" and no dynamic allocation is required.

There are few features that can be seen in this implementation that are not immediately related to the foregoing discussion:

- The dereference and indirection operators are `const` qualified, which allows the implementation class to overload on `const` in a natural manner.
- The single argument (conversion) constructor may be initialised using a class derived from `implementation` – this allows different specialisations of `implementation` to be selected at runtime[1]. This functionality is illustrated in the test case given in Listing 2.
- There is a two argument constructor that allows the user to provide custom copy and delete functionality.

Conclusion

The class template presented here - `arg::grin_ptr<>` - removes some of the repetitive work required when implementing Cheshire Cat classes. In addition it is able to support applications that are considerably more sophisticated (making use of polymorphic implementations and/or overloading on const) than the `phone_list` example considered here.

Alan Griffiths
<alan@octopull.demon.co.uk>

References

1. "shared_ptr class template", Greg Colvin et al., http://www.boost.org/libs/smart_ptr/shared_ptr.htm
2. "Ending with the grin", Alan Griffiths, http://www.octopull.demon.co.uk/arglib/TheGrin.html
3. 'Interpreting "Supporting the "Cheshire Cat" idiom"', Alan Griffiths & Allan Newton, *Overload 38*
4. "Constructing bullet-proof classes", John Carolan, *Proceedings: C++ at Work '89*, SIGS
5. *Design Patterns*, Gamma, Helm, Johnson, Vlissides ISBN 0-201-63361-2, Addison-Wesley
6. *Callbacks in C++ Using Template Functors*, Richard Hickey C ++ Gems ISBN 1 884842 37 2

1 This is a difference from the earlier implementationof `grin_ptr` – in that, if the user wanted to derive from `implementation`, it was necessary for `implementation` to derive from `arg::cloneable` and to declare the destructor virtual. I feel the current approach is more natural and extensible.

Listing 1 – grin_ptr.hpp

```
#ifndef INCLUDED_GRIN_PTR_HPP_ARG_20060308
#define INCLUDED_GRIN_PTR_HPP_ARG_20060308
#include <utility>

namespace arg
{
  template<typename Grin>
  class grin_ptr
  {
  public:

    /// Construct taking ownership of
    /// pointee.
    /// CheshireCat must be a complete type,
    /// Copyable and Destructable.

    template<typename CheshireCat>
    explicit grin_ptr(CheshireCat* pointee);

    /// Copy using copy function
    grin_ptr(const grin_ptr& rhs);

    /// Destroy using delete function
    ~grin_ptr() throw()  {
      copy_delete->do_delete(p); }

    /// Return contents (const)
    const Grin* get() const { return p; }

    /// Return contents (non-const)
    Grin* get() { return p; }

    /// Dereference op (const)
    const Grin* operator->()
     const { return p; }

    /// Dereference op (non-const)
    Grin* operator->() { return p; }

    /// Dereference op (const)
    const Grin& operator*()
     const { return *p; }

    /// Dereference op (non-const)
    Grin& operator*() { return *p; }

    /// Swaps contents with "with"
    void swap(grin_ptr& with) throw();
```

```
  ///
  grin_ptr& operator=(const grin_ptr& rhs);

  /// Pointers to deep copy and delete
  /// functions
  struct copy_delete_ptrs
  {
    typedef void (*delete_ftn)(Grin*);
    typedef Grin* (*copy_ftn)(const Grin*);

    copy_ftn      do_copy;
    delete_ftn    do_delete;
  };

  /// Allow user to specify copy and delete
  grin_ptr(Grin* pointee,
   copy_delete_ptrs* cdp)
    : p(pointee), copy_delete(cdp) {}

private:
  Grin*        p;
  copy_delete_ptrs* copy_delete;

  template<typename CheshireCat>
  static void my_delete_ftn(Grin* p);

  template<typename CheshireCat>
  static Grin* my_copy_ftn(const Grin* p);
};

template<typename Grin>
template<typename CheshireCat>
inline void
 grin_ptr<Grin>::my_delete_ftn(Grin* p)
{ delete static_cast<CheshireCat*>(p); }

template<typename Grin>
template<typename CheshireCat>
inline Grin*
 grin_ptr<Grin>::my_copy_ftn(const Grin* p)
{ return new CheshireCat(
  static_cast<const CheshireCat&>(*p)); }

template<typename Grin>
template<typename CheshireCat>
inline grin_ptr<Grin>::grin_ptr(
 CheshireCat* pointee)
 : p(pointee), copy_delete(0)
{
  void(sizeof(CheshireCat));
  static copy_delete_ptrs cdp =
    {
      &my_copy_ftn<CheshireCat>,
      &my_delete_ftn<CheshireCat>
    };

  copy_delete = &cdp;
}
```

```
  template<typename Grin>
  inline void grin_ptr<Grin>::swap(
   grin_ptr& with) throw()
  {
    std::swap(p, with.p);
    std::swap(copy_delete, with.copy_delete);
  }

  template<typename Grin>
  inline grin_ptr<Grin>::grin_ptr(
   const grin_ptr& rhs)
  :
    p(rhs.copy_delete->do_copy(rhs.p)),
    copy_delete(rhs.copy_delete)
  {
  }

  template<typename Grin>
  inline grin_ptr<Grin>&
  grin_ptr<Grin>::operator=(
   const grin_ptr& rhs)
  {
    grin_ptr<Grin>(rhs).swap(*this);
    return *this;
  }
}

namespace std
{
  template<class Grin>
  inline void swap(
    ::arg::grin_ptr<Grin>& lhs,
    ::arg::grin_ptr<Grin>& rhs) throw()
  {
    lhs.swap(rhs);
  }
}
#endif
```

Listing 2 – test for polymorphic implementation

```
struct base
{
  protected:
  base() {}
  base(const base&) {}
};

struct derived1 : base
{
  static int instances;

  derived1() { ++instances; }
  derived1(const derived1& src)
   : base(src) { ++instances; }
  ~derived1() { --instances; }
};
```

[continued at bottom of next page]

A Fistful Of Idioms

by Steve Love

Giving STL Iterators a Base Class

Some little while ago, I posted a question on the accu-prog-questions list regarding virtual template functions. The short answer I already knew – "it can't be done." The slightly longer answer was provided by Kevlin Henney – "it can be done if you know how."

A template function cannot be made virtual because, as Kevlin put it, *The dynamic binding you want ... is more significant than the polymorphism required to handle any type.* The virtual function table for the class would contain the address of the function normally, but a template function may never be instantiated, or may be instantiated multiple times. What's a poor compiler to do? Spit it out with a diagnostic, probably (which is exactly what mine did).

I thought that this could be overcome in a simple and idiomatic manner. I wasn't far off the mark; the solution provided by Kevlin is, in his words, a *collaboration of idioms.* I'll try to identify them as we go along.

The Good, the Bad and the Ugly

Consider a class to write to a file. The format of a given file is probably fixed (few files contain both fixed width and comma-separated values), but a more general solution will provide for several formats. Even better, a good solution will also provide for extensions; today you may want only CSV and fixed width, but tomorrow you may also require an ODBC data source. An obvious way to do this is provide an interface class, with pure virtual functions defining the interface to be implemented in derived classes. Each derived class has knowledge of the layout and delimiter requirements of the target.

```
class file_writer
{
public:
  // pure virtual functions define the
  // interface
};

class csv_writer : public file_writer
{
  // implement virtual functions according
  // to CSV format
};

class fixed_w_writer : public file_writer
{
  // implement virtual functions according to
  // fixed width format
};
```

The original requirement was for something along these lines. A record writer function would take a range of values to be written as a single record:

```
class file_writer
{
public:
    // ...
    template<typename Iterator>
    virtual void write
       (Iterator begin, Iterator end) = 0;
};
```

[continued from previous page]

```
int derived1::instances = 0;

struct derived2 : base
{
  static int instances;

  derived2() { ++instances; }
  derived2(const derived2& src)
   : base(src) { ++instances; }
  ~derived2() { --instances; }
};

int derived2::instances = 0;

void test_derived()
{
  assert(derived1::instances == 0);
  assert(derived2::instances == 0);

  {
    arg::grin_ptr<base> d1(new derived1);
    arg::grin_ptr<base> d2(new derived2);

    assert(derived1::instances == 1);
    assert(derived2::instances == 1);

    {
      // Check that copy works
      arg::grin_ptr<base> d(d1);
      assert(derived1::instances == 2);
      assert(derived2::instances == 1);

      // Check assignment from d1
      d = d1;
      assert(derived1::instances == 2);
      assert(derived2::instances == 1);

      // Check assignment from d2
      d = d2;
      assert(derived1::instances == 1);
      assert(derived2::instances == 2);
    }

    // Check destruction
    assert(derived1::instances == 1);
    assert(derived2::instances == 1);
  }

  // Check destruction
  assert(derived1::instances == 0);
  assert(derived2::instances == 0);
}
```

As already mentioned, this can't be done directly. Three possibilities immediately present themselves:

```
virtual void write
  (std::vector<std::string>::
    const_iterator begin,
  std::vector<std::string>::
    const_iterator end);

virtual void write
  (std::vector<std::string>container);
```

Neither of these is very pleasing; both tie you, as the client, into a specific container and contained type. The first has advantages over the second, in that you, the client, can pass only a partial range if you wish. The `write()` function itself doesn't have to change. Both suffer from having to specify the particular iterator to be used. For example, the function would need to be overloaded for a reverse iterator.

The third option is to pass each value in the container in turn to the `write()` function. This has the improvement over the previous solutions in that the container type is no longer an issue, but has other limitations, a burden on the client not least among them.

```
virtual void file_writer::write
 (const std::string & value) = 0;
// ...

std::vector<std::vector<std::string> > values;
file_writer * csv_file =
 new csv_file_writer (file_name);
// ...

for (std::vector<std::vector<std::string>
 >::iterator record = values.begin();
  record != values.end(); ++record)
{
  std::vector<std::string>::iterator i =
   record->begin();
  while (i != record->end()
  {
    csv_file->write (*i);
    ++i;
    if (i != record->end())
    {
      csv_file->write (",");
    }
  }
  csv_file->write ("\n");
}
```

This code would probably do the job, but is prone to error, slightly obfuscated (which could be helped with judicious use of using declarations), and the client is required to know the format of the output file – defeating the original object.

The clue to the solution is in the requirement. Ideally, we want to be able to write:

```
virtual void write (any_iterator begin,
 any_iterator end);
```

So, a good starting point is a class called `any_iterator`.

The Type With No Name

The reason for writing a function which operates on an iterator range as a template function, is so that it'll work with any iterator (provided the function itself uses a conformant interface). Given that fact, we need to be able to create an `any_iterator` from, well, any iterator. In this case, it is sufficient to be able to create one from any `Input Iterator`.

The concept of `Input Iterator` limits the scope of what we need to achieve in `any_iterator`. We need the following operations:

- default construction
- assignment/copy
- equality compare
- dereference
- member access
- pre increment
- post increment
- post increment and dereference[1]

On this basis, then, our `any_iterator` class will provide the following member functions:

- `any_iterator()`
- `any_iterator (const any_iterator &)`
- `operator= (const any_iterator &)`
- `operator== (const any_iterator &)`
- `operator!= (const any_iterator &)`
- `operator*()`
- `operator->()`
- `operator++()`
- `operator++(int)`

This provides our interface, but we have to decide, for example, what `operator*()` should return. We can specialise the entire class on its contained type (in our original requirement, `std::string`), or we can parameterise the class on the contained type:

```
template<typename Contained>
class any_iterator
{
    // ...
```

This latter method means that the signature for our `write()` function becomes

```
void write (any_iterator<std::string> begin,
            any_iterator<std::string> end)
```

which seems a reasonable compromse.

With this in mind, we can provide an implicit conversion from any `Input Iterator` (actually, from anything, but the restrictions will come later) using a member template function for a constructor.

```
template<typename Iterator>
any_iterator (const Iterator & rhs);
```

1 The "post-increment and dereference" isn't an operator in itself, but the expression `*iter++` uses both `operator*()` and `operator++(int)`. It merely requires that `operator++(int)` returns `*this` (usually as a `const` reference). Since some standard library functions use this idiom, `Input Iterator`s are required to support it. (ISO, 1998).

Painting Your Wagon

The real magic of the solution is expressed in a Pattern called "External Polymorphism" which is published as having the following intent:

Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them. [1]

Standard Library Iterators are expressly *not* related by inheritance; they are related by concept [2]. They do not have virtual functions, and cannot be treated polymorphically, in the late-binding sense. Our requirements for **any_iterator** are to treat **Iterators** as if they had a common ancestor. They could then be treated in the common manner described above.

```
class interface
{
public:
  // pure virtual interface declarations
};
template<typename ConcreteType>
class adaptor : public interface
{
public:
  // implementations of pure virtuals in
  // base class

private:
    ConcreteType adaptee;
};
```

The **adaptor** class has member functions which forward the call to the member function of **ConcreteType**. In this sense, this structure is similar in some respects to the Adaptor pattern [2], but it has differing motivations; it does not set out to convert the interface of **ConcreteType** in any way (athough it could), only to provide **ConcreteType** with polymorphic behaviour. The client class, using a pointer to interface, can use it as if it were a **ConcreteType**.

In our **any_iterator** class, **ConcreteType** is an iterator. We need to provide in the interface those functions which will allow us to provide an iterator-like interface in **any_iterator**. A start is the following:

```
class wrapper
{
public:
  virtual void next () = 0;
  virtual std::string & current () const = 0;
  virtual bool equal (
   const wrapper * rhs) const = 0;
  virtual void assign (
   const wrapper * rhs) = 0;
};
template<typename Iterator>
class adaptor : public wrapper
{
public:
  adaptor (const Iterator & rhs);
  virtual void next ()
    { ++adaptee; }
  virtual std::string & current () const
    { return *adaptee; }
  virtual bool equal (const wrapper * rhs)
    { return adaptee ==
      static_cast<adaptor<Iterator> *>
      (rhs)->adaptee; }
  virtual void assign (const wrapper * rhs)
    { adaptee =
     static_cast<adaptor<Iterator> *>
     (rhs)->adaptee; }

private:
    Iterator adaptee;
};
```

Note the addition of a conversion constructor in **adaptor**'s interface; that will become clear shortly. There are a couple of gotcha's here. The first one to go is the return from **current()**. This is the same contained type we already fixed in the **any_iterator** class by using a template parameter to the class. The easiest way to parameterise this one is to nest **interface** and **adaptor<>** inside **any_iterator**.

```
template<typename Contained>
class any_iterator
{
public:
    // ...

private:
  class wrapper
  {
  public:
    virtual void next () = 0;
    virtual Contained & current () const = 0;
    virtual bool equal (
     const wrapper * rhs) const = 0;
    virtual void assign (
     const wrapper * rhs) const = 0;
    };
    template<class Iterator>
    class adaptor : public wrapper
    {
        // ...
    };
};
```

The next problem is with **equal()**. The **rhs** pointer could have a different adaptee type to **this** – in our case, for example, a **list<>::iterator** when **this** one is a **vector<>::iterator**.

It is the polymorphic type which interests us, so we can make use of RTTI to determine consistency of type:

```
adaptor<Iterator> * tmp =
 dynamic_cast<adaptor<Iterator>*>(rhs)
return tmp && adaptee == tmp->adaptee;
```

The nature of using **dynamic_cast** on pointers means that **tmp** will be null if the conversion fails, i.e. the runtime type of **rhs** is different to **this**. To make this fail more noisily, we could have

`adaptor::equal()` take a reference, and make `tmp` a reference, thus forcing `dynamic_cast` to throw `std::bad_cast` on failure.

Note that `assign()` also suffers from this problem, but we'll address that one differently.

We now need access to this framework in the `any_iterator` class. The declarations are already nested; all that remains is to define such a thing.

```
template<typename Contained>
class any_iterator
{
public:
  // ...
private:
  class wrapper
  {
    // ...
  };
  template<typename Iterator>
  class adaptor : public wrapper
  {
    // ...
  };
  wrapper * body;
};
```

Now the member functions of `any_iterator` can operate on `body` which will forward all requests to `adaptee`.

```
const any_iterator<Contained>
 any_iterator<Contained>::operator++ (int)
{
  body->next();
  return *this;
}
bool any_iterator<Contained>::operator==
 (const any_iterator & rhs)
{
  return body->equal (rhs.body);
}
// ...
```

A Few Idioms More...

So far, so good. However, the functions we've created only work if we have an actual instance of `body`. Recall the converting constructor being a member template function. The template type of the Iterator is known at this point, and we can construct a new `adaptor` object accordingly:

```
template<typename Contained>
template<typename Iterator>
any_iterator<Contained>::any_iterator
 (const Iterator & rhs)
 : body (new adaptor<Iterator>(rhs))
{ }
```

The copy constructor is more of a problem; we have no way of identifying the Iterator type for `rhs->body`

```
any_iterator<Contained>::any_iterator
 (const any_iterator<Contained> & rhs)
{
    // ???
}
```

and so have no way of creating a new instance of it. This identifies a need for a new idiom in the `adaptor` class: a Virtual Copy Constructor [3, 4, 5].

A constructor for a class cannot be virtual. Normally, you know the concrete type of the object being created, and so this doesn't cause a problem. In some circumstances (here for example), we have only access to the base class type of the object we want to construct. It is meaningless to create an empty one, but perfectly reasonable to require a copy, or a *clone*. It forms part of the interface to the `wrapper` class

```
wrapper * wrapper::clone () const = 0;
```

and is implemented in `adaptor`:

```
template<typename Iterator>
wrapper * adaptor<Iterator>::clone () const
{
  return new adaptor<Iterator>(adaptee);
}
```

This uses the already given conversion from `Iterator` (the type of `adaptee`), which has been used before in the conversion constructor of `any_iterator`. The return from `clone()` could be covariant, i.e. return a pointer to the actual class rather than a pointer to its base class, but there is no benefit in this, since the target for the returned pointer is a pointer to the base class; it won't be used as part of an expression such as

```
Contained string_value = body->clone()
 ->current();
```

(Aside to users of Borland C++Builder 4/5 – I originally declared a copy constructor for adaptor thus:

```
adaptor (const adaptor<Iterator> & rhs);
```

but the entire class refused to compile. Removing the template type from the `rhs` parameter solved the problem legally, a language construct I had not previously encountered – but the diagnostics were not helpful in reaching this conclusion!)

This allows us to write a copy constructor for `any_iterator` as

```
template<typename Contained>
any_iterator<Contained>::any_iterator
 (const any_iterator & rhs)
 : body (rhs.body ? rhs.body->clone () : 0)
{ }
```

Having defined a copy constructor, we now need a copy assignment operator, which brings us to another idiom – Copy Before Release (CBR). Self assignment is not a problem in a class with only intrinsic data types (or those that provide the same copy semantics of intrinsics); but then, if only intrinsic data types are present, we can do without copy construction and copy assignment altogether, leaving the compiler to provide the necessary logic.

This actually leaves us at the Rule of Three – if you need any one of copy constructor, copy assignment operator, destructor, you generally need all three.[4, 6]

Since the data member of `any_iterator` is a pointer, we require a proper deep copy to be performed on assignment. We also

need to ensure that assignment to self does not occur. Finally, assignment should be exception safe, meaning it will operate correctly in the presence of an exception.

The idiom which expresses all these things is Copy Before Release [7]. At its most basic level, using the current example, it is implemented as

```
any_iterator<Contained> &
any_iterator<Contained>::operator=
 (const any_iterator<Contained> & rhs)
{
  wrapper * tmp = body->clone();
  delete body;
  body = tmp.body;
  return *this;
}
```

As you can see, the name given this idiom is apt. It turns out that this can be simplified using a suitable copy constructor. We've already defined the copy constructor for `any_iterator`, and so can use that. Further more, exception safety can be guaranteed by using `std::swap()`.This leads to a remarkably simple implementation [8]:

```
any_iterator<Contained> &
any_iterator<Contained>::swap
   (any_iterator<Contained> & rhs)
{
  std::swap (body, rhs.body);
  return *this;
}
any_iterator<Contained> &
 any_iterator<Contained>::operator=
 (const any_iterator<Contained> & rhs)
{
  any_iterator<Contained> temp(rhs));
  swap(temp);
  return *this;
}
```

The `swap()` member function merely swapping two pointers may look a little suspect, but the call to it provides the insight. We create a new copy of the `rhs` and swap it with `this`. When the call to `swap()` completes, the temporary object constructed in its arguments goes out of scope, taking its pointer with it. At that point, its `body` pointer is the memory which used to be attached to `this`, whereas `this->body` points at newly allocated memory containing a copy of `rhs`.

This safely handles null assignment, self assignment and exceptions (such as `std::bad_alloc`), as well as a successful copy. If an exception does occur, via the `any_iterator` copy constructor, then `this` will remain in its previous state, because the copy will not occur.

Conclusion

The whole premise of this technique revolves around being able to silently convert from a STL iterator to an `any_iterator`. In the wider sense [1], it can be used to superimpose polymorphic behaviour on other concrete types. This property of encouraging silent conversions is, rightly, regarded as dangerous (when C++ is described as a language which enables you to shoot yourself in the foot, this is one of the big shooters), but under specific conditions, such as those described here, it may be inescapable.

The circumstances under which it's considered dangerous are generally where the converted type appears as a parameter to a function; however, this is exactly the circumstance under which this idiom is intended to be used. Here's an example.

```
void csv_writer::write
  (const any_iterator<std::string> &begin,
  const any_iterator<std::string> & end)
{
  // ...
}
// ...
csv_writer cw;
std::vector<std::string> records;
// initialisation of vector with strings here
// using "ordinary" STL iterators from vector.
// It's also perfectly valid to use
// reverse_iterators or const_iterators
cw.write (records.begin(), records.end());

// similarly, we can use istream_iterators
// because the basis for any_iterator is the
// Input Iterator, which is modelled by
// istream_iterator
std::ifstream strm (file_name);
cw.write
 (std::istream_iterator<std::string>(strm),
  std::istream_iterator<std::string> ());
```

This extract demonstrates the use of `any_iterator`. The intention is that client code never uses – indeed, probably doesn't even need to know about – the `any_iterator` type directly, but is able to use it transparently because it exhibits properties to allow this.

Steve Love

Acknowledgements

My thanks to Kevlin Henney, who provided more than just "significant input" to both this article and the code that backs it, and Nigel Dickens, who provided further code and text reviews.

References

1 Cleeland, (1996) *External Polymorphism*, http://siesta.cs.wustl.edu/~cleeland/papers/External-Polymorphism/External-Polymorphism.html

2 Gamma, Helm, Johnson and Vlissides (1995), *Design Patterns*, Addison Wesley

3 Henney, Kevlin (1999) Overload, "Coping With Copying in C++ - Clone Alone" August 1999

4 Cline (1991) *The C++ FAQ*, http://www.cs.bham.ac.uk/~jdm/CPP/index.html

5 Koenig, Andrew and Barbara Moo (1997) *Ruminations on C++*, Addison Wesley/AT&T

6 Coplien, James O (1992) *Advanced C++ Programming Styles & Idioms*, Addison Wesley

7 Henney, Kevlin (1998) *C++ Report*, "Creating Stable Assignments".

8 Sutter, Herb (1999) *Exceptional C++*, Addison Wesley

9 Leone, Sergio Spaghetti Western Director

C++ Best Practice – Designing Header Files

by Alan Griffiths

C++, and its close relatives C and C++/CLI, have a compilation model that was invented way back in the middle of the last century – a directive to insert text into the translation unit from (at least in the typical case) a header file. This has the advantage of simplicity – and to run on 1970's computers compilers had to be simpler than they do today. However, there are other considerations and it is not clear that the same trade-off would be made if inventing a language today (certainly more recently designed languages like Java and C# import a compiled version of referenced code).

A consequence of this language design decision is that the responsibility for dealing with these "other considerations" shifts from the language and compiler writer to the developers using the language. Naturally, experienced C++ developers do not think about all these as they work, any more than an experienced driver thinks about where she places her hands and feet when turning a corner. But just as many drivers get into bad habits (and cut corners), many programmers fail to observe best practice at all times, so it is worth reviewing these factors occasionally.

Considerations for Designing C++ Header Files

1. The effect of including a header file should be deterministic (and not be dependent upon the context in which it is included).
2. Including a header file should be idempotent (including it several times should have the same effect as including it once).
3. A header file should have a coherent purpose (and not have unnecessary or surprising effects).
4. A header file should have low coupling (and not introduce excessive dependencies on other headers).

It should come as no surprise that this list is analogous to the design criteria for other elements of software – functions, classes, components, subsystems, etc. It includes well known techniques for making design elements easier to use, reuse and reason about. There are exceptions: for example, functions that may produce a different result on each invocation – **`std::strtok`** is one – but not all design is good design. (Not even in the standard library.)

If all header files met these criteria, then it would make the life of developers easier. However, because the language doesn't forbid ignoring these principles there are header files that don't satisfy them. One needs to look no further than the standard library and **`<cassert>`** to find an example of a header file that breaks the first two criteria. Not only does the effect of this header depend upon whether the preprocessor macro NDEBUG has been defined[1], but it is also designed to be included multiple times in a translation unit:

```
#undef NDEBUG
#include <cassert>
void assert_tested_here() { assert(true); }

#define NDEBUG
#include <cassert>
void assert_untested_here() { assert(false); }
```

1 It is also worth noting that one should be very wary of code using **`assert`** in header files as it may expand differently in different translation units. (Which would break the "One Definition Rule".)

Other examples exist too: Boost.preprocessor [1] makes use of a technique referred to as "File Iteration" (repeatedly `#including` a file under the control of preprocessor macros to generate families of macro expansions). Although this technique is undeniably useful, the circumstances where it is appropriate are rare – and the vast majority of headers do meet (or should meet) the conditions given above.

Writing Deterministic, Idempotent Headers

Most header files however are designed to be independent of when or how many times they are included. Lacking direct language support, developers have come up with a range of idioms and conventions[2] to achieve this. These methods relate only to the structure of the header which means that they are simple, effective and easy to follow:

- Only include header files at file scope. Including header files within a function or namespace scope is possible but it isn't idiomatic (and won't work unless the header is designed to make this possible).

```
// Do this...
#include "someheader.hpp"

// Not this
namespace bad_idea
{
    #include "someheader.hpp"
}
```

- The *Include Guard Idiom* is to surround the body of the include file by an **`#ifndef`** block and define the corresponding macro in the body. The body of the header, therefore, has no effect after the the first inclusion (but this does not avoid the file being reopened and parsed by the compiler's preprocessor[3]).

```
// Like this
#ifndef INCLUDED_EXAMPLE_HPP_ARG_20060303
#define INCLUDED_EXAMPLE_HPP_ARG_20060303
...
#endif
```

  Note that one needs some mechanism for ensuring the header guards are unique. (Some development environments will generate guaranteed unique guards for you – here I've just used a combination of the header name, my initials and the date.)
- Create *Self Contained Headers* that do not rely on the presence of macros, declarations or definitions being available at the point of inclusion. Another way to look at this is that they should be compilable on their own. One common way to ensure headers

2 This is not as effective as support within the language: not only does it require developers to know the idioms but also compilers can't be optimised on this asumption as they must support headers for which it is not true.

3 There is a technique for avoiding this compile-time cost – *External Include Guards*: each **`#include`** can be surrounded by a **`#ifndef`**...**`#define`**...**`#endif`** block akin to the include guard idiom mentioned below. It does work (I've reduced a 12 hour build cycle to 4 hours this way) but it is painful to use because it is verbose, and it is necessary to ensure that the guard macros are chosen consistently. Use it only in extreme need and, before using it, check your compiler for optimisations that would make it unnecessary.

are self-contained is to make them the first include in the corresponding implementation file.

- *Don't use using directives – especially in the global namespace (and particularly* `using namespace std;`*).* Users of the header file may be surprised and inconvenienced if names are introduced into scopes they think they control – this is particularly problematic with using directives at file scope as the set of names introduced depends upon which other headers are included, and name lookup in client code always has the potential to search the global namespace.

Compiler suppliers also like to add value and, on the basis that almost all header files are deterministic and idempotent, have used this assumption in attempts to reduce compilation time:

- There are compilers (like gcc) that recognise the Include Guard Idiom and don't process such header files (again) if the corresponding macro is defined. As the cost of opening files and parsing them is often a significant component of the compile time the saving can be significant.
- There are compilers (like Microsoft's Visual C++) that implement a language extension – `#pragma once` – that instructs the compiler to only process a header file the first time it is encountered. As the cost of opening files and parsing them is often a significant component of the compile time the saving can be significant.
- Some compilers (like Microsoft's Visual C++) allow headers to be "pre-compiled" into an intermediate form that allows the compiler to quickly reload the state it reaches after processing a series of headers. This may be effective if there is a common series of headers at the start of multiple translation units – opinions differ about the frequency of this occurring (if it is infrequent there may be a net cost of pre-compiled headers).
- There have also been "C++" compilers (such as IBM's VisualAge C++) that have abandoned the inclusion model of compilation – but that makes them significantly non-compliant and, to the best of my knowledge, this approach has been abandoned.

With the appropriate co-operation from the developer any, or all, of these can be effective at reducing build times. However, only the first of these is a portable approach in that it doesn't have the potential to change the meaning of the code. (If the code in a header file does have a different effect if included a second time then `#pragma once` changes its meaning; if files sharing a "precompiled header" actually have different sequences of includes the meaning is changed; and, with no separate translation units, VisualAge C++ interpreted `static` definitions and the anonymous namespace in non-standard ways.)

Writing Coherent Headers

It is much harder to meet this objective by rote mechanical techniques than the two considered above. This is because a single design element may map to several C++ language elements (for example, the `<map>` header provides not only the `std::map<>` class template, but also a number of template functions such as the corresponding `operator==` and a class template for `std::map<>::iterator`. It takes all of these elements to support the single "ordered associative container" concept embodied in this header. (Of course, this header also includes a parallel family of templates implementing "multimap" – it is less clear that this is part of the same concept.)

There have been various attempts to give simple rules for the design of coherent headers:

- "*Put each class in its own header file*." I frequently encounter this rule in "coding standards" but while classes and headers are both units of design they are not at the same level of abstraction. The example given above of a container and its iterators shows that this rule is simplistic. However, the weaker (but harder to explain) rule "if you have two classes in the same header then question whether they support the same abstraction" is probably useful.
- "*Can the purpose of the header be summarised in a simple phrase without using 'and' or 'or'?*" This is an application of a common test for coherence in a design. The problem with this test is that one needs to "get it" before it is useful: is "represent the household pets" a single concept? Or is "represents the household cats, dogs and budgerigars" multiple concepts? Given an appropriate context either might be true.

Of course, there are common mistakes that can be avoided:

- *Never create a header called* `utils.hpp` *(or variations like* `utility.h`*).* It is too easy for a later developer to add things to this header instead of crafting a more appropriate one for their purpose. (On one project I worked on `utility.h` got so cluttered that one of my colleagues created a `bits-and-bobs.h` header "to make it easier to find things". It didn't work.)

Writing Decoupled Headers

Occasionally one encounters adherents of a "pretend C++ is Java" style of header writing – just put the implementation code inline in the class definition. They will cite advantages of this techniques: you only code interfaces once (no copying of function signatures to the implementation file), there are fewer files to keep track of, and it is immediately apparent how a function works. (There is a further advantage that I've never seen mentioned – it is impossible to write headers with circular dependencies in this style[4].) Listing 1 shows a header written in this style.

In the listing I've highlighted the pieces of code that are of no value to translation units that use the header file. This is of little account when there are few users of the `telephone_list` class (for example, when the header is first being written and there is only the test harness to consider[5]). However, having implementation code in a header distracts from defining the contract with the client code.

Further, most C++ projects are considerably larger than this and, if this approach were applied to the entire codebase, the cost of compiling everything in one big translation unit would be prohibitive. Even on a smaller scale exposing implementation code can also have an adverse effect on the responsiveness of the build process: compiling the implementation of a function or member-function in every translation unit that includes the header is wasteful; in addition, implementation code often requires more context than interfaces (e.g. the definitions of classes being used rather than declarations of them); and, finally, the implementation

4 This is not true in Java, but C++ is not Java.

5 I suspect that I'm not the only one to apply "test driven development" to C++ by initially writing some pieces of new code in a header. If you decide to try this approach remember: refactoring the code to separate out the implementation afterwards is part of the work you need to do. This is not a case where "you ain't gonna need it" applies

Listing 1

```
// allinline.hpp - implementation hiding
// example.
#ifndef INCLUDED_ALLINLINE_HPP_ARG_20060303
#define INCLUDED_ALLINLINE_HPP_ARG_20060303
#include <string>
#include <utility>
#include <map>
#include <algorithm>
#include <ctype.h>

namespace allinline
{
  /** Example of implementing a telephone list
    * using an inline implementation.
  */
  class telephone_list
  {
  public:
    /** Create a telephone list.
    * @param    name    The name of the list.
    */
    telephone_list(const std::string& name)
     : name(name), dictionary() {}
    /** Get the list's name.
    * @return   the list's name.
    */
    std::string get_name()
     const { return name; }
    /** Get a person's phone number.
    * @param    person  The person's name
    * (must be an exact match)
    * @return   pair of success flag and (if
    * success) number.
    */
    std::pair<bool, std::string> get_number(
     const std::string& person) const {
     dictionary_t::const_iterator
     i = dictionary.find(person);
     return(i != dictionary.end()) ?
        std::make_pair(true, (*i).second) :
        std::make_pair(false, std::string());
    }

    /** Add an entry. If an entry already
    * exists for this person it is
    * overwritten.
    *   @param  name    The person's name
    *   @param  number  The person's number
    */
    telephone_list& add_entry(
     const std::string& name,
     const std::string& number) {
       std::string nn(name);
       std::transform(nn.begin(), nn.end(),
        nn.begin(), &tolower);
       dictionary[nn] = number;
       return *this;
    }

  private:
    typedef std::map<std::string,
            std::string> dictionary_t;
            std::string  name;
            dictionary_t dictionary;

    telephone_list(const telephone_list& rhs);
    telephone_list& operator=(
     const telephone_list& rhs);
  };
}
#endif
```

is more likely to change than the interface and trigger a mass recompilation of the client code.

Hence:

- *Don't put implementation code in headers*. If we apply this rule to our example we arrive at something like Listing 2[6]. Once again, I've highlighted stuff that is of no interest to the client code.
- *Don't include unnecessary headers*. It may seem obvious, but the easiest thing that developers can do to reduce the number of header files being included is not to include them when they are not needed. It takes only one unnecessary header in each file to have an enormous effect on compilation times: not only do the original files each add an extra file, but so do each of the added files, and the files they add, and... – the only reason that this doesn't go on forever is that eventually some of these includes are duplicated and, because of their include guards, these duplicates do not include anything the second or subsequent times they are included.

When is a Header File Include "Necessary"?

An examination of Listing 2 shows that I've highlighted three includes – two of these can be removed without preventing the header compiling while the third is not used in the interface to client code – it is an "implementation detail" that has leaked. There are also two headers (not highlighted) that are needed to compile the header itself and also include definitions used in the interface to client code. The two sidebars ("Cheshire Cat" and "Interface Class") show two alternative design techniques that permit removing these includes from the header file. Of course, they are still required by the implementation file and need to be moved there.

Deciding whether it is necessary to include a header file isn't quite as simple as removing it and asking "does it compile?" – a trial and error approach to eliminating header files can generate both false positives and false negatives: positives where a header appears unnecessary incorrectly (because it is also being included indirectly by another header – where it may be unnecessary), and negatives where a header is incorrectly being considered necessary (because it indirectly includes the header that is actually needed). There really is no shortcut that avoids understanding which definitions and declarations a header introduces and which of these are being used – and this is easier when headers have a coherent function.

6 This example may look familiar to some of you – in Overload 66 Mark Radford and I used a similar example in our article "Separating Interface and Implementation in C++". I've stolen it (and the "Cheshire Cat" and "Interface Class" sidebars) because it addresses the current theme.

Cheshire Cat

- A private "representation" class is written that embodies the same functionality and interface as the naïve class – however, unlike the naïve version, this is defined and implemented entirely within the implementation file. The public interface of the class published in the header is unchanged, but the private implementation details are reduced to a single member variable that points to an instance of the "representation" class – each of its member functions forwards to the corresponding function of the "representation" class.
- The term "Cheshire Cat" is an old one, coined by John Carollan over a decade ago [2]. Sadly it seems to have disappeared from use in contemporary C++ literature. It appears described as a special case of the "Bridge" pattern in *Design Patterns* [3], but the name "Cheshire Cat" is not mentioned. Herb Sutter [4] discusses it under the name "Pimpl idiom", but considers it only from the perspective of its use in reducing physical dependencies. It has also been called "Compilation Firewall".
- Cheshire Cat requires "boilerplate" code in the form of forwarding functions that are tedious to write and (if the compiler fails to optimise them away) can introduce a slight performance hit. It also requires care with the copy semantics (although it is possible to factor this out into a smart pointer [5]). As the relationship between the public and implementation classes is not explicit it can cause maintenance issues.

```
// cheshire_cat.hpp Cheshire Cat -
// implementation hiding example
#ifndef INCLUDED_CHESHIRE_CAT_HPP_ARG_20060303
#define INCLUDED_CHESHIRE_CAT_HPP_ARG_20060303
#include <string>
#include <utility>
namespace cheshire_cat
{
  class telephone_list
  {

  public:
    telephone_list(const std::string& name);

    ~telephone_list();
    std::string get_name() const;
    std::pair<bool, std::string>
    get_number(const std::string& person)
                const;
    telephone_list&
    add_entry(const std::string& name,
              const std::string& number);

  private:
    class telephone_list_implementation;
    telephone_list_implementation* rep;
    telephone_list(const telephone_list& rhs);
    telephone_list& operator=
                  (const telephone_list& rhs);
  };
}
#endif
```

Listing 2

```
// naive.hpp - implementation hiding
// example.
#ifndef INCLUDED_NAIVE_HPP_ARG_20060303
#define INCLUDED_NAIVE_HPP_ARG_20060303
#include <string>
#include <utility>
#include <map>
#include <algorithm>
#include <ctype.h>

namespace naive
{
  /** Telephone list. Example of implementing a
  *telephone list using a naive implementation.
  */

  class telephone_list
  {
  public:
    /** Create a telephone list.
    * @param    name    The name of the list.
    */
    telephone_list(const std::string& name);
    /** Get the list's name.
    * @return   the list's name.
    */
    std::string get_name() const;
    /** Get a person's phone number.
     * @param    person  The person's name
     * (must be an exact match)
     * @return   pair of success flag and (if
     * success) number.
     */
    std::pair<bool, std::string>
    get_number(const std::string& person)
     const;
    /** Add an entry. If an entry already
    * exists for this person it is overwritten.
    *
    *   @param  name    The person's name
    *   @param  number  The person's number
    */
    telephone_list&
    add_entry(const std::string& name,
              const std::string& number);

  private:
    typedef std::map<std::string,
                     std::string> dictionary_t;
    std::string  name;
    dictionary_t dictionary;
    telephone_list(const telephone_list& rhs);
    telephone_list& operator=
     (const telephone_list& rhs);
  };
}
#endif
```

Listing 3

```
#ifndef INCLUDED_LISTING3_HPP_ARG_20060303
#define INCLUDED_LISTING3_HPP_ARG_20060303

#include "location.hpp"
#include <ostream>

namespace listing3
{
  class traveller
  {
  public:
    ...
    location current_location() const;
    void list_itinary(std::ostream& out) const;
  };
}
#endif
```

There are also other ways to avoid including headers. Frequently, a header file will bring in a class (or class template) definition when all that is needed is a declaration. Consider Listing 3 where the header file `location.hpp` is included whereas all that is needed is the statement "class location;". A similar approach can be taken with `std::ostream` – but with a `typedef` in the standard library one cannot do it oneself: one cannot say `typedef basic_ostream <char, char_traits<char> > ostream;` without first declaring `template<...> class basic_ostream;` and `template<...> class char_traits;` and doing this is problematic because the bit I've shown as "..." is implementation defined.

The designers of the standard library did realise that people would want to declare the input and output stream classes without including the definitions, and allowed for this by providing a header

[continued at bottom of next page]

Listing 4

```
#ifndef INCLUDED_LISTING4_HPP_ARG_20060303
#define INCLUDED_LISTING4_HPP_ARG_20060303

#include <iosfwd>

namespace listing4
{
  class location;

  class traveller
  {
  public:
    ...
    location current_location() const;
    void list_itinary(std::ostream& out) const;
  };
}
#endif
```

Interface Class

All member data is removed from the naïve class and all member functions are made pure virtual. In the implementation file a derived class is defined and implements these member functions. The derived class is not used directly by client code, which sees only a pointer to the public class.

This is described in some detail in Mark Radford's "C++ Interface Classes – An Introduction" [6].

Conceptually the Interface Class idiom is the simplest of those we consider. However, it may be necessary to provide an additional component and interface in order to create instances. Interface Classes, being abstract, can not be instantiated by the client. If a derived "implementation" class implements the pure virtual member functions of the Interface Class, then the client can create instances of that class. (But making the implementation class publicly visible re-introduces noise.) Alternatively, if the implementation class is provided with the Interface Class and (presumably) buried in an implementation file, then provision of an additional instantiation mechanism – e.g. a factory function – is necessary. This is shown as a static `create` function in the corresponding sidebar.

As objects are dynamically allocated and accessed via pointers this solution requires the client code to manage the object lifetime. This is not a handicap where the domain understanding implies objects are to be managed by a smart pointer (or handle) but it may be significant in some cases.

```
// interface_class.hpp - implementation
// hiding example.
#ifndef INC_INTERFACE_CLASS_HPP_ARG_20060303
#define INC_INTERFACE_CLASS_HPP_ARG_20060303
#include <string>
#include <utility>

namespace interface_class
{
  class telephone_list
  {
  public:
    static telephone_list* create(
     const std::string& name);
    virtual ~telephone_list()   {}
    virtual std::string get_name() const = 0;
    virtual std::pair<bool, std::string>
    get_number(const std::string& person)
                const = 0;
    virtual telephone_list&
    add_entry(const std::string& name,
              const std::string& number) = 0;
  protected:
    telephone_list()   {}
    telephone_list(const telephone_list& rhs)
      {}
  private:
    telephone_list& operator=
     (const telephone_list& rhs);
  };
}
#endif
```

Visiting Alice

by Phil Bass

"The time has come," the Walrus said,
"To talk of many things:
Of tuples, trees and composites;
Of visitors and kings."[1]

Welcome

"Good morning, everyone, and welcome to the Wonderland Social Club annual treasure hunt. I am the Walrus." (*coo-coo coo-choo*) "Well, not a walrus, but I am quite long in the tooth." (*groan*)

"This year the clues are all in trees. On each clue sheet there's a clue to an item of treasure. Some clue sheets also contain two further clues, which lead to more clue sheets. With each treasure clue there is an indication of the value of the treasure at that location."

"You have until 6 o'clock this evening to find as much treasure as you can. The team with the most valuable hoard of treasure will be the winner. The first clue is outside in the garden. See you back here in the Carpenter's Arms at 6 o'clock. Good luck everybody!"

Planning the Route

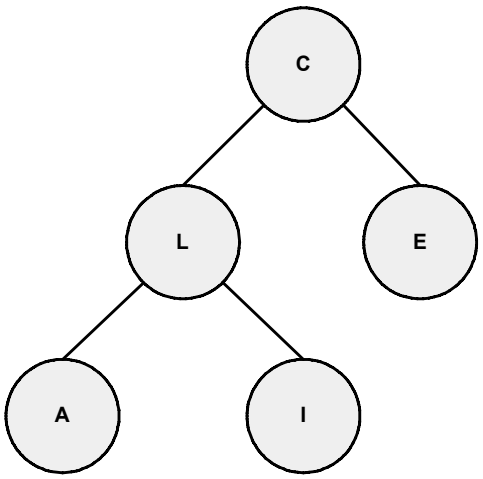
There were three teams: four trainee nurses called the Pre-Tenders, three employees of the Royal Mail called the Post Men and two publicans called the Inn Keepers. The Pre-Tenders decided to do the easy clues first; the Post Men chose to visit the nearest places first; and the Inn Keepers settled for finding the most valuable treasure first.

Overload readers will have spotted immediately that the treasure hunters' problem involves the traversal of an abstract binary tree. The Walrus had drawn the tree on a sheet of paper so that he could refer to it when he was adding up the scores at the end of the day. And, as it turned out, the Pre-Tenders would visit the treasure locations in pre-order sequence, the Post Men in post-order sequence and the Inn Keepers in in-order sequence.

Encoding the Problem

Bill, a nerdy-looking youth with thick oyster-shell glasses had spotted this, too. He was a C# programmer and was often to be seen in the corner of the bar with a beer and a lap-top. To him the treasure hunters' tree seemed to be a set of dynamically constructed, garbage collected, polymorphic objects. "It's a binary tree", he said to his best mate, Ben. "Yes", agreed Ben, but Ben's mental imagery was very different. "Nested structs", said Ben.

Bill looked at him blankly for a moment, decided Ben must have been joking and replied with an ironic, "Yeah, right". But Ben was a C++ programmer and he wasn't joking. "No, really" said Ben, pulling out his own lap-top, "Look, I'll show you what I mean".




```
struct L
{
  char a, i;
};

struct C
{
  L    l;
  char e;
};
```

Listing 1 - A simple tree as nested structs

1 After Lewis Caroll's "The Walrus and the Carpenter". By the way, he lied about the kings.

[continued from previous page]

file for the purpose - **<iosfwd>**. While this doesn't avoid including any header it is much simpler than **<ostream>** and with its aid we can write Listing 4.

Conclusion

Designing C++ header files is like many tasks in developing computer software – if done badly it can cause major problems. I hope that I've shown that, given the right techniques and idioms plus an understanding of the issues, doing it well isn't so terribly hard.

Alan Griffiths
<alan@octopull.demon.co.uk>

References

1. "An Introduction to Preprocessor Metaprogramming", David Abrahams and Aleksey Gurtovoy, http://boost-consulting.com/tmpbook/preprocessor.html
2 "Constructing bullet-proof classes", John Carrolan, Proceeding C++ at Work - SIGS
3 Gamma, Helm, Johnson & Vlissides. "Design Patterns", Addison-Wesley, 1995
4 Herb Sutter. Exceptional C++, Addison-Wesley, 2000
5 "To Grin Again", Alan Griffiths, *Overload 72*.
6 "C++ Interface Classes – An Introduction", Mark Radford, *Overload 62*

See also "Separating Interface and Implementation in C++", Alan Griffiths and Mark Radford, Overload 66

```
class Component
{
public:
  virtual ~Component() {}
  virtual void operation() = 0;
  virtual void push_back(Component*) = 0;
  typedef std::vector<Component*>
   ::iterator Iterator;
  virtual Iterator begin() = 0;
  virtual Iterator   end() = 0;
};

class Leaf : public Component
{
public:
  Leaf(int v) : leaf_value(v) {}
  int value() {return leaf_value;}

private:
  virtual void operation() { /* . . . */ }
  virtual void push_back(Component*)
   {throw "Can't add to Leaf!";}
  virtual Iterator begin()
   {return children.begin();}
  virtual Iterator   end()
   {return children.end();}
  int leaf_value;
  std::vector<Component*> children;
  // always empty
};

class Composite : public Component
{
private:
  virtual void operation() { /* . . . */ }
  virtual void push_back(Component* c)
   {children.push_back(c);}
  virtual Iterator begin()
    {return children.begin();}
  virtual Iterator   end()
    {return children.end();}

    std::vector<Component*> children;
};
```

Listing 2 - Classes for the Composite Pattern in C++

Ben drew a tree with just five nodes, A, L, I, C and E, as shown in Listing 1. He defined classes for the root node, C, and the other non-leaf node L. For the leaf nodes he just used **char**s.

Bill was not impressed at all. "I thought you were a C++ programmer", he retorted. "That's C code. Where are the classes? Where are the methods? What's that public data doing there?" "This just captures the *structure* of the tree", Ben explained. "I've used a **char** as a place-holder for the treasure clues and the value of the treasure."

Bill wasn't convinced. Inserting his *Design Patterns* CD into his PC he brought up the Composite structure diagram and typed in an improved version of Ben's classes. "Even I can write better

```
int main()
{
  C tree = /* ... */;
  visit<in_order>(tree, Write(cout));
  return 0;
}
```

Listing 3 - A generic tree visit function

C++ than that", Bill scoffed. Referring to the example in the *Design Patterns* book, Bill wrote the code shown in Listing 2. Now it was Ben's turn to be unimpressed. Bill's code was more than three times longer (32 non-blank lines to 9), it had more types in more complex relationships, it brought up apparently irrelevant implementation issues (such as using vectors or lists and how to implement **Leaf::push_back()**) and it was less efficient in time and space (particularly if the tree nodes were allocated on the heap). But Ben was more interested in other things.

"OK", said Ben, in an uncharacteristically conciliatory tone, "but your tree has to be built at run time and it gives special status to **Component::operation()**. What if the tree structure is fixed at compile time? And what if we want to capture the tree structure, but don't know what operations will be performed on the nodes?" Bill's face fell. He could see what was coming. Any second now Ben was going to say something about those confounded C++ templates.

Two of the trainee nurses from the Pre-Tenders team had been powdering their noses. As they came out of the Ladies and left the pub to start the treasure hunt Bill's face brightened momentarily. But his brief reverie was broken by Ben's words. "We should be able to write a function template that applies an arbitrary function to each node in a tree. Any tree. Something like this." And Ben typed in the code in Listing 3.

"Let's stick with our 5-node tree and let's not worry about how it's created, for now. Write is a function object that writes information about a node to an **ostream**. And **visit** is a function template that walks the tree and calls a function (**Write** in this case) for each node. The **in_order** template parameter is a policy class that specifies in-order traversal."

```
template<typename Traversal, typename Node,
typename Op>

void visit(Node& root, Op op)
{
  typedef typename begin<Node>::type first;
  typedef typename   end<Node>::type last;
  typedef typename Traversal
   ::template apply<Node>::type pvp;

  children<first, pvp>::template
   visit_each<Traversal>(root, op);
  op(root);
  children<pvp, last>::template
   visit_each<Traversal>(root, op);
}
```

Listing 4 - The generalised visit algorithm

```
template<typename C, typename M, M C::*>
struct next;

template<>
struct next<C, L, &C::l>
{
    static int L::* const value;
};
int C::* const next<C, L, &C::l>::value
 = &C::e;
```

Listing 5 - The compile-time analogue of incrementing an iterator

The Generalised Visit Algorithm

Thinking out loud, Ben said: "When we visit a node we need to apply the given function to the node itself and each of its child nodes. The parent defines a natural order on the children and we'll use that to visit each child in turn. The traversal policy defines when to visit the parent (before the children, after the children or somewhere in between). It's like having a parent that's an STL container of children and the traversal policy provides an iterator into the container – except that it all happens at compile time. So the algorithm is:

- visit the children in the range [first, pvp)
- visit the parent
- visit the children in the range [pvp, last)

where pvp is the 'parent visit position' iterator obtained from the traversal policy."

Ben's code is shown in Listing 4.

At first this looked more like hieroglyphics than source code to Bill. He could see Ben's "iterators" `first`, `last` and `pvp`, but they were *types* and real iterators are *objects*. He could see function calls that seemed to correspond to the three steps of the visit algorithm, too, but it was far from clear how the `visit_each` functions would work. The "iterators" were being used to instantiate the `children` class template instead of being passed to the `visit_each` function, so how could the function step through the children as the algorithm requires?

Compile-Time Iterators

Ben was well aware that the code needed some explanation. "Compile time algorithms work with constants and types, which are both immutable", he went on. "There's no such thing as a compile-time variable. Instead of incrementing an iterator we must create a new iterator pointing to the next element in the container. For example, the root node in our 5-node tree has two children: l and e. We could think of using a pointer-to-member-of-C as an iterator. Then an iterator to the first child of C would point to `C::l` and we can imagine incrementing that iterator to point to `C::e`. But C++ doesn't have a pointer-to-member-of-C type – it only has pointer-to-member-of-C-of-type-L (`L C::*`); and that can't be incremented because it would change type in the process (becoming `int C::*`)."

Bill wasn't sure if he understood this, but he wasn't going to admit it, so he let Ben continue. "Instead we can define a meta-

```
template<typename C, typename M,
   M C::*> struct member_iterator;
template<typename D,
   typename B> struct base_iterator;
```

Listing 6 - Member and base class iterators.

```
template<typename First, typename Last>
struct children
{
  template<typename Traversal, typename Node,
   typename Op>
  static
  void visit_each(Node& node, Op op)
  {
    visit<Traversal>
     (bind<First>::function(node), op);
    typedef typename next<First>::type Next;
    children<Next,Last>::template
     visit_each<Traversal>(node, op);
  }
};

template<typename Iter>
struct children<Iter, Iter>
{
  template<typename Traversal, typename Node,
   typename Op>
  static
  void visit_each(Node&, Op) {}
};
```

Listing 7 - The `visit_each<>` function.

```
template<typename Iter> struct bind;

template<typename C, typename M,
 M C::* member>
struct bind< member_iterator<C, M, member> >
{
  typedef member_iterator<C, M, member> Iter;
  typedef typename  child<Iter>::type Child;
  typedef typename parent<Iter>::type Parent;
  static Child& function(Parent& parent)
   {return parent.*member;}
};

template<typename D, typename B>
struct bind< base_iterator<D,B> >
{
  typedef base_iterator<D,B> Iter;
  typedef typename  child<Iter>::type Child;
  typedef typename parent<Iter>::type Parent;
  static Child& function(Parent& parent)
   {return parent;}
};
```

Listing 8 - Simulated partial specialisations of the bind function.

function that takes a compile-time iterator as a template parameter and generates a new compile-time iterator pointing to the next element. Something like this." Ben produced listing 5.

"We pass in `&C::l` (as a template parameter) and we get out `&C::e` (as a class static value)."

"But that's hideous", said Bill. "It is pretty ugly", admitted Ben. "And it only works for child nodes that are accessible data members of the parent node, too. That's very limiting. But the `visit` function doesn't use pointer-to-member values as iterators – it uses types, and that's much more flexible."

Ben was enjoying himself. "Suppose L was a base class of C instead of a member", he enthused. "We can't use a pointer-to-member value to identify a base class, but we can encode a pointer-to-member value as a type." (In fact, the `next<>` meta-function in Listing 5 does just that.) "And then we can use types as iterators to base classes *and* data members. It's probably worth declaring class templates for these two families of types: `member_iterator<>` points to a node that's a member of its parent and `base_iterator<>` points to a node that's a base class sub-object of its parent."

"I'll refer to these collectively as 'child iterators'. The `begin<>` and `end<>` meta-functions can return classes instantiated from these templates for the `visit<>` function to use. Those meta-functions are the compile-time analogue of the `begin()` and `end()` functions provided by STL containers."

"OK", said Bill slowly, still far from convinced, "We can't increment these iterators, but we still have to compare them and dereference them, right? How do we do that?"

Iterating Through the Children

"That's a good question", replied Ben. "We need to see how the `visit_each<>` function works to answer that."

"As you can see, there are no loop constructs. There can't be because loops require a loop variable and there are no compile-time variables. So we use recursion instead. The idea is to visit the first child (by calling the `visit<>` function) and then call `visit_each<>` recursively to visit all the remaining children."

Ben paused to take another gulp of his Black Sheep bitter and Bill seized the opportunity to ask why `visit_each<>` is a static function of a template class. "I'll come to that in a minute", Bill responded, "but for now you just need to know that the primary `children<>` template defines the general case and the specialisation provides the terminating condition for the recursion."

Ben pondered for a moment before continuing. "If `First` is a `member_iterator<>` the `visit_each<>` function can retrieve a pointer-to-member value from it, which must be bound to an object using the `.*` or `->*` operator before we can access the data member it refers to. We can't store an object reference or pointer in the iterator because it's a *compile-time* iterator and references and pointers are run-time entities – they don't have a value at compile time. And that means we can't dereference a `member_iterator<>` – it doesn't hold enough information. So, the `visit_each<>` function must bind a compile-time iterator to a (run-time) reference to the parent node, which produces a reference to the child node pointed to by the iterator. And, of course, if we have to do this for `member_iterator<>`s we should do the same for other child iterators such as `base_iterator<>`s."

"Is that clear?", asked Ben. "Errm, I think so", said Bill, who wasn't used to thinking about the interaction of compile-time operations with the run-time environment. "So, a child iterator is a type, not an object; it can't be incremented; and it can't be dereferenced. That's a weird sort of iterator!" Ben agreed, "Weird, yes, but it's still an iterator in my book".

Bill was studying the `visit_each<>` function and another question occurred to him: "`bind<>` seems to be a class template with a nested function. Why isn't it just a template function?" "Ah", said Ben, "that's so that we can define separate bind functions for member-iterators and base-iterators. If C++ supported partial specialisation of function templates we could define partial specialisations for each of the two iterator families; but it doesn't, so I've just put a static function in a class template and defined partial specialisations of the class template."

Hardly pausing for breath Ben continued his commentary. "Here, the `parent<>` and `child<>` meta-functions just extract the type of the parent and child nodes from the child iterator." (See Appendix 1.) "The `member_iterator<>` specialisation uses the `.*` operator to bind a pointer-to-member to the parent; the `base_iterator<>` specialisation just uses the implicit conversion from derived class reference to base class reference." (Although Ben didn't think to mention it, he could have used the Boost `enable_if<>` templates to define separate overloads of the `bind` function instead.)

Ben leant back in his chair with a satisfied smile. "The rest is easy", he said. "The `visit_each<>` function uses the `next<>` meta-function to generate an iterator to the next child node and calls itself to visit the remaining children (if any). Eventually, the two iterators become equal and the `children<Iter,Iter>` specialisation comes into play. At that point an empty `visit_each<>` function is instantiated and the recursion terminates."

Bill felt cheated. "So we don't *compare* the iterators, either", he said tetchily. "At least, not explicitly." "Well, no, I suppose not", Ben replied with an air of superiority, "but that's how it is with compile-time algorithms."

Epilogue

"Glad to see everyone back on time", said the Walrus cheerily. "Now, let's see how you all got on." And with that he collected the clue sheets and sat down to work out what the teams had scored. Some 20 minutes later he stood up again and gravely announced that he had a problem – all three teams had the same score! "So I'll offer a bonus point to the first team to find Tweedledum and Tweedledee." One of the Pre-Tenders gave a little squeal, waved one hand in the air and pointed excitedly with the other in the direction of Bill and Ben. "It's them", she cried. And everyone could see she was right. They looked like two peas in a pod and they were arguing like schoolboys.

"Compile-time is better", said Ben. "Nah, run-time", insisted Bill. "Compile-time." "Run-time." ...

Phil Bass
phil@stoneymanor.demon.co.uk

Full code listings provided in appendices.

Appendix 1 – Visit.hpp

The following code implements the compile-time visit algorithm itself.

```
struct nil {};
template<typename Iter> struct parent;
template<typename Iter> struct child;
template<typename Iter> struct bind;
template<typename Iter> struct next;
template<typename C, typename M, M C::*> struct member_iterator;
template<typename C, typename M, M C::* member>
struct parent< member_iterator<C,M,member> >
{
  typedef C type;
};

template<typename C, typename M, M C::* member>
struct child< member_iterator<C,M,member> >
{
  typedef M type;
};

template<typename C, typename M, M C::* member>
struct bind< member_iterator<C, M, member> >
{
  typedef member_iterator<C, M, member> Iter;
  typedef typename  child<Iter>::type Child;
  typedef typename parent<Iter>::type Parent;
  static Child& function(Parent& parent) {return parent.*member;}
};

template<typename D, typename B> struct base_iterator;
template<typename D, typename B>
struct parent< base_iterator<D,B> >
{
  typedef D type;
};

template<typename D, typename B>
struct child< base_iterator<D,B> >
{
  typedef B type;
};

template<typename D, typename B>
struct bind< base_iterator<D,B> >
{
  typedef base_iterator<D,B> Iter;
  typedef typename  child<Iter>::type Child;
  typedef typename parent<Iter>::type Parent;
  static Child& function(Parent& parent) {return parent;}
};

template<typename T> struct begin { typedef nil type; };
template<typename T> struct end { typedef nil type; };
template<typename First, typename Last> struct children;
template<typename Traversal, typename Node, typename Op>
void visit(Node& root, Op op)
{
  typedef typename begin<Node>::type first;
```

```
    typedef typename   end<Node>::type last;
    typedef typename Traversal::template apply<Node>::type pvp;
    children<first, pvp> ::template visit_each<Traversal>(root, op);
    op(root);
    children<pvp, last> ::template visit_each<Traversal>(root, op);
  }
  template<typename First, typename Last>
  struct children
  {
    template<typename Traversal, typename Node, typename Op>
    static
    void visit_each(Node& node, Op op)
    {
      visit<Traversal> (bind<First>::function(node), op);
      typedef typename next<First>::type Next;
      children<Next,Last> ::template visit_each<Traversal>(node, op);
    }
  };
  template<typename Iter>
  struct children<Iter, Iter>
  {
    template<typename Traversal, typename Node, typename Op>
    static
    void visit_each(Node&, Op) {}
  };
```

Appendix 2 - Visiting Alice (with children as nested structs)

A program using the visit algorithm described here needs to provide some information about the structure of the tree whose nodes are to be visited. This is done by defining suitable child-iterator types, the `next<Iter>` meta-function, the `begin<Node>` meta-function and specialisations of the traversal order policy templates. The following code shows the definitions for the ALICE tree used in the main text.

```
  struct L
  {
    char a, i;
  };
  typedef member_iterator<L, char, &L::a> L_a_iterator;
  typedef member_iterator<L, char, &L::i>  L_i_iterator;
  template<> struct next<L_a_iterator> { typedef L_i_iterator type; };
  template<> struct next<L_i_iterator> { typedef nil          type; };
  template<> struct begin<L> { typedef L_a_iterator type; };
  struct C
  {
    L    l;
    char e;
  };
  typedef member_iterator<C, L,    &C::l> C_l_iterator;
  typedef member_iterator<C, char, &C::e> C_e_iterator;
  template<> struct next<C_l_iterator> { typedef C_e_iterator type; };
  template<> struct next<C_e_iterator> { typedef nil type; };
  template<> struct begin<C> { typedef C_l_iterator type; };
  struct in_order
  {
    template<typename T> struct apply { typedef nil type; };
  };
  template<> struct in_order::apply<C> { typedef C_e_iterator type; };
  template<> struct in_order::apply<L> { typedef L_i_iterator type; };
```

The traversal policy is in the form of a meta-function class (a class with a nested class template). This gives the flexibility of template template parameters while still allowing the policy to be used by compile-time algorithms operating on types.

Appendix 3 - Visiting Alice (with a base class as a child)

The case of the ALICE tree where L is a base class of C is only slightly different. The root node, C, needs a constructor and the child-iterator pointing to node L becomes a `base_iterator<>`. The following code shows the support required for node C (most of which is the same as for the example in which L is a member of C).

```
struct C : L
{
  C(const L& l, char e_) : L(l), e(e_) {}
  char e;
};

typedef   base_iterator<C, L> C_l_iterator;
typedef member_iterator<C, char, &C::e> C_e_iterator;

template<> struct next<C_l_iterator> { typedef C_e_iterator type; };
template<> struct next<C_e_iterator> { typedef nil type; };
template<> struct begin<C> { typedef C_l_iterator type; };
template<> struct in_order::apply<C> { typedef C_e_iterator type; };
```

For compile-time trees defined using tuples and inheritance it is possible to provide the child iterators in a library. For example, the ALICE tree could be defined as:

```
#include <boost/tuple.hpp>
using boost::tuple;
typedef tuple<char,char> L;
typedef tuple<L,char> C;
C tree = C( L('A','I'), 'E' );
```

Since Boost tuples are implemented using inheritance it is easy to provide predefined base-iterators and meta-functions supporting the `visit<>` algorithm. In this case, no information about the structure of the tree needs to be provided in the client code; visiting Alice comes for free.