overload|80

August 2007
ISSN: 1354-3172
www.accu.org

The PfA Papers: From the Top
Kevlin Henney

He Sells Shell Scripts to Intersect Sets
Thomas Guest

Release Mode Debugging
Roger Orr

auto_value: Transfer Semantics for Value Types
Richard Harris

OVERLOAD 80

August 2007

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 81 should be submitted to the editor by 1st September 2007 and for Overload 82 by 1st November 2007.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

Consensus

It should be obvious that the process of agreeing effective standards must be dependant on attaining consensus

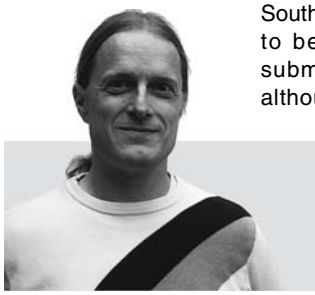
It should be obvious that the process of agreeing effective standards must be dependant on attaining consensus.Over recent editorials I've commented on the increasingly concerning activities occurring in the ISO PAS standardisation process. One of the cases I've been following (the C++/CLI standard) came to an early resolution as a result of the efforts of Herb Sutter (Microsoft's liaison to the C++ community, and a major contributor to the ECMA C++/CLI work). Despite his investment in C++/CLI after seeing the reaction of the C++ community he managed to steer a path through the rules and regulations of ISO and stop the progress of this purportive standard at the ballot resolution stage. Initially JTC1 [ISO's Joint Technical Committee 1] refused ECMA's attempt to withdraw C++/CLI, but after lining up support for this withdrawal from a number of national bodies a second attempt to cancel the Ballot Resolution Meeting was allowed.

I'm sure that as developers many of us have experience of projects that only progress in something resembling an orderly manner as a result of the heroic efforts of individuals. Indeed, it is part of the description of CMM Level 1. Despite it being common this really is no way to run a project. (Sooner or later any organisation will run out of heroes.) It is also no way to run a standardisation process!

In Overload 75 I wrote the following:

> Every organisation develops a culture over the course of time that reflects the way it tries to work. And ISO is no exception to this. Most of its standards are of interest to a minority of those on the decision making panels (not surprising really, there are a lot of standards and very limited resources to pursue them). A consequence of this is that national bodies with no interest in a particular standard try to keep out of the way by automatically voting 'approve' to anything that comes up for a vote. For traditional standards this is justified on both the tit-for-tat principle that others will do the same for standards that do interest them and on the assumption that those national bodies forming the working group have been diligent.
>
> In organisational terms the fast-track is a new thing and the existing culture of 'approve' by default is still operating. However, for a PAS submission there may be no national bodies interested in the standard – with the consequence that neither the tit-for-tat principle nor the presumption of diligence need apply. The effect of this can be quite alarming.

Given these concerns I was very pleased when a paper written by one of the national standards bodies was brought to my attention:



> South Africa is concerned about what seems to be a growing number of standards submitted under the PAS process that, although publically [sic] available, do not seem to have any measure of regional or even national consensus. These therefore tend not to have been referred to any of the JTC 1 sub-committees, and have obviously not been discussed at [sub-committee] level.
>
> Our experience is that the result of this is then a round of intense lobbying by various other stakeholders for us to vote negatively on the PAS. Often these other groups take the trouble to compile a list of contradictions that are also widely distributed in order to justify the request for the negative vote.
>
> A recent example is the proposed PAS on Open XML/ODF.
>
> It is our opinion that the submission of such 'standards' directly to JTC 1 via the PAS route, where the standard has not been discussed within the relevant SC, was never the intention of the PAS System. The fact that some consortium has published a document that they refer to as a standard does not automatically imply that it has any sort of widespread industry acceptance. The fact that the publisher might claim international usage or acceptance is not longer a valid reason in these days of large multinationals, and the SABS [South African Bureau of Standards] has previously been approached by local branches of multinationals to vote in support of such PAS submissions, even if we have no local industry involvement or membership in the appropriate JTC 1 SC.
>
> As result of this, South Africa will tend to vote negatively on all future PAS submissions to JTC 1 where they have not been appropriate SC. We would like to ensure that proper consideration be given to the PAS by technical experts. If the standard is indeed well known within the industry then this process might be very short.
>
> This will be a change from our previous tendency to 'abstain' where we had no direct knowledge of the submission.

I may be reading more into the timing than is strictly warranted, but a further standards body (this time DIN – representing Germany) has been comparing the various standardisation processes that occur under the ISO auspices. It has the following to say about the fast-track process:

> In case of the Fast-Track process, the proposer does not need to prepare any of the required information for a NP (certainly he is free to do so to support the proposal). National Bodies have no influence over the acceptance or rejection of a Fast-Track proposal at this stage as there is no formal 'acceptance' ballot foreseen. Only when 'perceived contradictions' are identified can they raise concerns. NBs/SCs have to 'accept' the additional workload brought in by a Fast-Track proposal, which may also have an unforeseen impact on their prioritised work plan (Art 6.3.1.4). In addition there is no check if there are enough P-Members committed to work on the proposal and to thoroughly review the proposed specification.

Alan Griffiths is an independent software developer who has been using "Agile Methods" since before they were called "Agile", has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

> Even if NBs can clearly demonstrate that the submitted specification duplicates an existing standard, the Fast-Track proposer can still request to enter the technical ballot phase. During the technical ballot phase, a formal rejection is only allowed based on technical arguments (see Annex G18: "If a national body finds the DIS unacceptable, it shall vote negatively and state the technical reasons." and "We disapprove for the technical reasons stated"). JTC1 and its NBs are therefore not able to reject standard proposals based on arguments like business needs, stability, maturity of the technology, lack of independent and interoperable implementations, etc.

The other purportive standard I've been reporting on is that of OOXML – an ECMA document format standard whose claimed scope is supporting documents produced by MS Office (there are contentions that this standard fails to address this scope adequately). There are now accounts [Groklaw] appearing on the internet of national body committees being 'packed' with delegates with the intention of forcing a 'yes' vote on approving this standard. Vis:

> We've seen now reports from Italy and Portugal of what some are describing as a kind of ballot-stuffing on the part of Microsoft and supporters to get Ecma-376 approved as an ISO standard. Trust me when I tell you that you haven't heard the half of it yet. I feel safe in saying that you will never hear the phrase 'fast tracking' again, without remembering what you are about to read.

It has never been a surprise that powerful organisations exert their power in their own interests. But many may be surprised that there is sufficient value in standardising a document format for these forces to come into play. There are two reasons – there are some (usually government related) markets that will insist on conformance to standards (because standards promote competition and other good stuff) and Microsoft doesn't want to lose these markets, and also because competitors implementing the alternative international standard (ODF) can use the interoperability this enables to facilitate their own marketing.

Somewhere in the midst of all this manoeuvring the ideal of gaining consensus has been replaced with that of competition. Consensus could serve everyone – competition serves only one winner.

Reference

[Groklaw] http://www.groklaw.net/article.php?story=2007071812280798

Letter to the Editor

Dear Alan,

I was delighted to read 'C++ Unit Test Frameworks – a Comparison' by Chris Main in Overload 78. Obviously I am mostly pleased because Aeryn (http://www.aeryn.co.uk/) came out pretty much on top compared to other well known frameworks like Boost and against Peter Sommerlad's new CUTE framework. I am particularly grateful to Chris Main for his comments on the readability of the Aeryn user guide and for highlighting that Aeryn is missing at least one set of test condition macros. I am currently in the process of incorporating these test condition macros into Aeryn and they will be available in the next release (I should also point out that this functionality is already available with Boost and CUTE).

Shortly before Overload 78 was published I had all but completed my own article comparing Aeryn, CUTE and FRUCTOSE and demonstrating how Aeryn is the only testing framework you need. Obviously to have a similar, and mostly likely more balanced article, come from an independent author who is not known to me is much better. It would have been even better to have a comparison with CPPUnit. This is something Jim Hyslop and I spoke about a few years ago, but never got around too.

Chris Main pointed out in his article that there have been quite a few C++ unit testing frameworks in Overload over the past few months and a comment was made during the 'writing for publication' BoF session at 2007 ACCU conference that people would like a break from them for a while. This is a shame as I am reasonably close to completing my first introductory article for Aeryn. It's difficult to believe that in the years I have been developing Aeryn I have never written an article just about Aeryn. Now I feel the opportunity has passed, which is entirely my own fault, so I will only be publishing this article on the Aeryn website.

I'd also like to bring to the attention of Overload readers an Aeryn related project being developed by Steve Love and a new Aeryn tool I'm intending to develop.

I've often wanted to develop an NUnit style GUI for Aeryn. I've looked at a number of solutions using shared libraries in C++ and using managed C++ to enable integration into a C# GUI. Steve Love, as one of Aeryn's most avid users, has been having the same idea and is developing NAeryn (http://naeryn.tigris.org/), a C# GUI which utilises Aeryn's existing binary interface and a custom XML report.

At the 2007 ACCU conference, Peter Sommerlad demonstrated a very nifty Eclipse plug-in for his CUTE framework and prior to that a (admittedly tongue in cheek) suggestion was made by Michael Baker that he might use Aeryn if it was integrated into Eclipse. Therefore I am intending to develop an Aeryn plug-in for Eclipse. Hopefully this will encourage more open source projects to move across from CPPUnit to Aeryn.

All that remains is for me to reiterate my thanks and appreciation to Chris Main for all the positive and constructive points he made about Aeryn in his article.

Thanks Chris!

Paul Grenyer
paul.grenyer@gmail.com

He Sells Shell Scripts to Intersect Sets

Graphical User Interfaces are in great demand but for some tasks there are better tools. Thomas Guest demonstrates the capabilities of command shells.

Introduction

The Unix command shell contains a lot of what I like in a programming environment: it's dynamic, high-level, interpreted, flexible, succinct. It's even reasonably portable now that bash seems to have become the shell of choice. Although there's much about shell scripting I don't like, on many occasions it turns out to be the best tool for the job.

In this article we shall demonstrate how simple shell scripts can be used to implement sets, providing one line recipes for set creation, set union, set intersection and more. Having explored the power of the Unix shell we'll consider its limitations, before finally discussing the more general lessons we can learn from the Unix tools.

An example: Apache server logs

As an example, let's suppose we want to analyse sets of IP addresses contained in a couple of Apache HTTP Server [Apache] access logs: `access_log1` and `access_log2`. Each log file contains many thousands of lines which look something like this:

```
65.214.44.29 - - [25/Jun/2007:00:03:21 +0000] ...
74.6.87.40 - - [25/Jun/2007:00:03:24 +0000] ...
65.214.44.29 - - [25/Jun/2007:00:03:24 +0000] ...
74.6.86.212 - - [25/Jun/2007:00:03:36 +0000] ...
...
```

We can **`cut`** this file down to leave just the IP address at the start of each line. **`cut`** is a simple tool which we'll be using again later, and here we're passing it options **`-f1`** to select the first field from each line and **`-d" "`** to use the space character as a field separator.

```
$ cut -f1 -d" " access_log1
65.214.44.29
74.6.87.40
65.214.44.29
74.6.86.212
...
```

Set creation

The output from this command is likely to be full of duplicates. Regular site visitors typically hit the web server a few times; web spiders and robots are much more hungry. To obtain the **sets** of unique IP addresses contained in each log file, we could do this:

```
$ cut -f1 -d" " access_log1 | sort | uniq > IP1
$ cut -f1 -d" " access_log2 | sort | uniq > IP2
```

Here **`cut`** picks out the IP addresses, **`sort`** orders the results, **`uniq`** eliminates duplicates, and we've redirected the output into files IP1 and IP2. By the way, we could have eliminated a link from the pipeline using the **`-u`** option to sort. The Unix shell tools aren't entirely orthogonal!

```
$ cut -f1 -d" " access_log1 | sort -u > IP1
```

The resulting sets are ordered – a set implementation which should be familiar to C++ programmers. The IP addresses will be lexicographically rather than numerically ordered, since we went with the **`sort`** defaults. This means that, for example, **`122.152.128.10`** appears before **`58.167.213.128`** because **`1`** alphabetically precedes **`5`**. With a little more effort, we could probably persuade **`sort`** to yield a numeric ordering (no, **`sort -n`** isn't good enough).

Multiset creation

If instead we wanted a **multiset** – that is, a set in which elements may appear more than once, we could count the number of times items are repeated in the sorted output using the **`-c`** option to **`uniq`**.

```
$ cut -f1 -d" " access_log1 | sort | uniq -c
   8 12.153.20.132
   2 12.217.178.11
  14 12.30.66.226
   1 122.152.128.49
  ...
```

Here, each IP address is prefixed by the number of times it occurred in the log file, so our multiset contains **`12.153.20.132`** 8 times, etc. This will be useful later when we discuss intersection operations.

Set union

Let's assume we've followed the steps above and IP1 and IP2 contain the set of IP addresses in the two access logs. Forming the **`union`** of these sets is simple.

```
$ sort -m IP1 IP2 | uniq > IP1_union_IP2
```

The **`-m`** (merge) option to **`sort`** is purely for efficiency and the result would be equally correct without it. Since the inputs are already sorted, we can just merge them together, line by line. For C++ users, it's the difference between the **`std::sort`** and **`std::merge`** algorithms.

Thomas Guest Thomas is an enthusiastic and experienced programmer. He has developed software for everything from embedded devices to clustered servers. Contact him at thomas.guest@gmail.com and his website is http://www.wordaligned.org

you can create your own customised shell working environment and port it from platform to platform just by checking it out

Set intersection

The best recipe I've come up with for creating the **intersection** of the sets IP1 and IP2 isn't quite as simple. Here's how it works. We form the multiset union of IP1 and IP2, then filter it to leave just the elements which appear twice in this multiset.

```
$ sort -m IP1 IP2 | uniq -c | grep "^ *2" | \
  tr -s " " | cut -f3 -d" " > IP1_intersection_IP2
```

Let's unpick this pipeline. First, **sort -m IP1 IP2 | uniq -c** generates the multiset of IP addresses in IP1 and IP2. Since IP1 and IP2 are sets and therefore *individually* contain no repeats, the resulting multiset looks something like this:

```
$ sort -m IP1 IP2 | uniq -c
   1 12.30.66.226
   1 122.152.128.10
   2 122.152.128.49
   1 122.152.129.54
   ...
```

Each line in the output starts with a count which *must* be either 1 or 2. Lines starting with **2** correspond to IP addresses common to both files – and these are the IP addresses which form the intersection of IP1 and IP2. The other lines, the ones starting with **1**, are the IP addresses in just one of IP1 or IP2.

We then use **grep** to pick out lines starting with any number of spaces followed by a **2**. Next **tr -s " "** squeezes repeated spaces from each line, making the output suitable for use with **cut** using the space character as a field delimiter. Finally **cut** itself extracts the column we want (the one with the IP address).

Set symmetric difference

The same recipe with a simple tweak finds the **set symmetric difference** between IP1 and IP2 (the IP addresses in just one of IP1 and IP2 that is). All we need to do is change the **grep** pattern to include a **1** instead of a **2**. The magic of the shell command history allows us to hit the up arrow key – ⇧ – and edit the previous command directly; we don't even have to type it all in again.

```
$ sort -m IP1 IP2 | uniq -c | grep "^ *1" | \
  tr -s " " | cut -f3 -d" " > IP1_symmetric_diff_IP2
```

Set subtraction

What about the elements in IP1 but not IP2? We can just intersect **IP1** and **IP1_symmetric_diff_IP2**. Again, we can use the command shell history to recall and adapt the previous command.

```
$ sort -m IP1 IP1_symmetric_diff_IP2 | uniq -c |
grep "^ *2" | \
  tr -s " " | cut -f3 -d" " > IP1_subtract_IP2
```

More set operations

One of the nice things about set operations is there aren't many of them. We've already covered the important ones, and these can easily be extended. Try and work out what set operations are going on in the the command history shown below.

```
$ diff -q S1 S2
$ head -1 S1
$ sort -m S1 S2 S3 S4 S5 | uniq
$ sort -m S1 S2 S3 | uniq -c | grep -c "^ *3"
$ sort -m S1 S2 | uniq -c | grep "^ *2" | tr -s " "
| cut -f3 -d" " | diff S1 -
$ sort -m S1 S2 | uniq -c | grep -c "^ *2"
$ tail -1 S2
$ wc -l S1
```

As a hint, the answers in lexicographical order are:

- are two sets the same?
- are two sets disjoint?
- first element of a set
- how big is the intersection of three sets?
- how many elements in a set?
- is a subset of?
- last element of a set
- unite five sets

Extending the toolset

The command shell is a powerful, dynamic and extensible programming environment. Even these simple one-line scripts can be stored as functions which can be sourced when a new shell is started; you can add command-line help to them, you can find them using tab-completion, you can keep them in your source control system. In this way you can create your own customised shell working environment and port it from platform to platform just by checking it out [Guest06a].

A script's got to know its limitations

Apache server logs are no more and no less than line oriented text. Each record in the log is terminated by a newline character, and each field within each record is delimited in an obvious way: by brackets, spaces, hyphens, whatever – who needs XML? This is the kind of format shell scripts handle well. Conversely, anything more complicated, XML for example, or records which span multiple lines, is likely to push the shell tools too far. Maybe **awk** could cope, but I don't think many people bother learning **awk** these days: it's better to use one of the popular high-level languages when basic shell commands won't do.

Shell scripts tend not to fail safely. For example, the following command is meant to clear out files in a temporary directory:

a compact suite of **orthogonal tools**, each with its own responsibility, which cooperate using **simple interfaces**

```
# Don't try this at home!
$ rm -rf $TEMP_WORK_DIR/*
```

You can imagine what happens if `TEMP_WORK_DIR` has not been set. In general, the Unix commands build on a couple of dangerous assumptions: that programmers know what they are doing; and that the show must go on – by which I mean that, given malformed input, a shell script will not throw an exception. The IP filters we discussed in this article work quite happily with any old text file as input – if it wasn't an Apache http server log, the only indication of failure may well be smaller sets than expected.

I'll admit that I personally avoid writing any shell scripts much longer than the ones shown here. As with Makefiles, I admire and respect the technology but I'd rather have someone else deal with the details. The **bash** manual may be brief to a fault, but I've yet to get to grips with its finer details. Sometimes it's just too subtle.

On the subject of details, earlier in this article I said that by default **sort** uses lexicographical ordering, which isn't perhaps the ordering we'd prefer for IP addresses; and I also said that a numeric **sort -n** wouldn't do the job either: IP addresses aren't really numbers, they're dot separated number quartets. You *can* use **sort** to place IP addresses in a more natural order, but the command you'll need is anything but natural.

```
# Natural ordering of IP addresses
$ sort -t. +0n -1n +1n -2n +2n -3n +3n IP
```

If you want to know how this works you'll have to read the manual. The code, on its own, is unreadable [Guest06b]. If you don't know where the manual is, just open a shell window and type **man**. If the output from this command doesn't help, try **man man**, and if you don't know how to open a shell window, I'm surprised you're even reading this sentence!

Conclusion

Modern graphical development environments tend to hide the shell and the command line, probably with good reason, and I don't suppose this article will persuade anyone they're worth hunting out. And yet the Unix shell embodies so much that is modern and, I suspect, future, best practice.

For me, it's not just what the shell tools can do, it's the example they set. Look again at some of the recipes presented in this article and you'll see container operations without explicit loops. You'll see flexible and generic algorithms. You'll see functional programming. You'll see programs which can parallel-process data without a thread or a mutex in sight; no chance of shared memory corruption or race conditions here. The original design of the shell tools may have become somewhat polluted – we've already seen that **sort** does some of what **uniq** can do – but I think the intent shines through as clearly as ever: we have a compact suite of orthogonal tools, each with its own responsibility, which cooperate using simple interfaces. We would do well to emulate this model in our own software designs. ■

Credits

I would like to thank the Overload editorial team for their help with this article.

References

[Apache]: http://httpd.apache.org/

[Guest06a]: http://blog.wordaligned.org/articles/2006/09/07/personal-version-control

[Guest06b]: http://blog.wordaligned.org/articles/2006/08/06/readable-code

The PfA Papers: From the Top

A characteristic of patterns is that experienced developers often experience a moment of recognition upon reading the write up. Sometimes the write up isn't even needed...

The PARAMETERISE FROM ABOVE pattern (PfA) has acquired a certain notoriety on accu-general. I first described it by this name a number of years ago as a general approach that reduced dependencies on global assumptions, whether constants or environmental features. The guideline covers a family of different techniques that all share an inversion of responsibilities and dependencies in a design.

A by-product of PfA – and also a commonly cited motivation for it – is the reduction of SINGLETONs and other globals in a body of code. It is this characteristic that is most often associated with the pattern, as the following quote indicates [Radford2006]:

> One perspective on PARAMETERISE FROM ABOVE is that it is the alter-ego of SINGLETON (and other approaches involving globally accessible objects). SINGLETON is a dysfunctional pattern – one that transforms the design context for the worse, rather than for the better. PARAMETERISE FROM ABOVE is a pattern that is 'out there', but for which (to the best of my knowledge) there is (so far) no formal write up.

The closing sentence also hints at why this pattern has become notorious: in spite of frequent citation of the pattern and encouragement to do so, I have not yet documented it. I do not propose to break this tradition just yet, but in response to a recent posting of mine [Henney2007] – and as an antidote to Teedy Deigh's seasonal foolishness [Deigh2007] – I have decided to document some of PfA's history and uses. So let's take it from the top.

Parameterisation from the top

The general formulation and naming of the pattern has its origins at a particular client. It became a phrase and form of phrase I found myself commonly using in making design suggestions at this particular site. This much I remembered, and vaguely recalled it being around 2002 – there are presentations I have dated 2002 where the PfA term is used explicitly. What I had forgotten, until recently stumbling across it again, was that I had actually documented the basic idea in the appendix of a report for the client the previous year (July 2001, to be precise).

The client's system comprised a number of applications built from a common set of code. The report made a number of specific recommendations for repartitioning the utility packages and features in this system. The appendix in question was entitled 'Parameterisation from the Top', the full text of which is as follows:

> One of the recurring issues in the use of constants in the [...] system is the hard coding of runtime-related names and constraint values, such as path names, queue names, event names, and so on. Although named constants have generally been used to avoid hardwiring the literal constants into the code, this has only made the coupling problem more obvious: the whole system is tightly coupled to application-level concepts. This means that even the smallest change to any of these values in future will result in a total system rebuild, i.e. all of the executables and libraries from which the system is comprised. It also means that code cannot be reused simply in other systems and that testing is always limited by these values, where others might be more appropriate for a test harness.
>
> Such configuration values are not global, and therefore the use of global magic names has caused this problem. As the term configuration suggests, these are runtime configuration values that should supplied to each, and therefore by each, application (i.e. process). These values should be supplied explicitly by the initialising applications to the objects and classes that need them (either as constructor arguments or as template parameters, as appropriate). In other words, the practice being described here is that the parameterisation of a system should come from its top level downwards, and not from its bottom level upwards.
>
> The first step is to move all use of global constants to the top layer of the system, and ensure that constants are passed down the layers as necessary. The next step is to factor out the commonality that exists between many of the applications, creating a simple application framework: a base class that handles command-line parsing, common initialisation, error handling and program execution. Global constants can then be migrated into the relevant derived classes. A final step would be to support runtime specification of many of these parameters through the application framework – configuration files, registry entries, command-line arguments, environment variables, etc.

Although this note is missing much of the customer context and system detail, the general idea comes through. In terms of PfA, it is pretty much all there, but with a particular focus on constants, and the idea of a more general design approach hinted at the end. The (dis)association with SINGLETON and the associated with STRATEGY objects, CONTEXT OBJECTs, MOCK OBJECTs, etc., became more prominent the following year, as did the more relative – rather than absolute naming – of the practice. ■

References

[Deigh2007] Teedy Deigh, 'A Practical Form of OO Layering', *Overload 78*, April 2007, http://accu.org/index.php/journals/1327

[Henney2007] Kevlin Henney, 'Parameterisation from the Top', accu-general, 2nd June 2007

[Radford2006] Mark Radford, 'C++ Interface Classes: Strengthening Encapsulation', *Overload 76*, December 2006, http://accu.org/index.php/journals/1329

Kevlin Henney is a long-standing member of ACCU, joining before it actually was ACCU and contributing to Overload when it was numbered in single digits. He recently co-authored two volumes in the Pattern-Oriented Software Architecture series, *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. Kevlin can be contacted at kevlin@curbralan.com.

Release Mode Debugging

Although we try not to put bugs into software, sometimes we must track them down to remove them. Roger Orr considers the difficulties when they can only be seen in release builds.

Introduction

Most programmers are familiar with debugging; although the amount of time spent debugging depends on the programmer as well as the environment and the problem domain. However, in a number of different segments of the I.T. industry, there is a dichotomy between 'Debug' and 'Release' builds. This is most often related to development in a compilable language rather than one which is interpreted.

The phrasing implies you debug using the 'Debug' build and then release software built with the 'Release' build. I personally don't like this split – my own preference is to have a single build – but in particular this nomenclature is misleading.

Experience shows that it's not this simple – not all the bugs are removed during development, some will be discovered using the release build. Unfortunately the phrasing (and some of the tool chains) make it harder than it needs to be to debug any problems found in the release version of the product.

I will re-examine the difference between the two builds and then provide some examples of things that can be done to make it easier to find and fix faults in the 'Release' build. The examples are for C/C++ but similar concerns exist in build environments for other languages.

What is the difference between a 'Debug' and a 'Release' build?

The idea behind the split builds is fairly sound for all but the most agile of development processes. There are two main target groups for software – the developers and the users – whose use of the software places different requirements on it. For example, during software development it is usually preferable to stop the program as soon as possible after a problem is detected to make the job of detecting – and removing – the cause of the fault as easy as possible. By contrast, most users of the program would prefer that some attempt is made to recover from the fault and to ensure no valuable data is lost.

A second difference is the level of access that should be granted to the two teams. The developers usually have full access to the original source code, and can be allowed access to the internals of the program at runtime. The users are probably not interested in the internal workings of the program and, for commercial programs, there may be strong reasons to restrict such access to try and retain intellectual property rights.

Hence many of the tool chains provide two (or sometimes more than two) targets with different characteristics. A 'Debug' build is designed for developers and typically:

- contains full symbolic information for the binary files
- has not been optimised
- provides additional tracing and debugging functionality
- often contains checks for memory use (stack, heap or both)

A 'Release' build is designed for users and typically:

- is smaller in size and built with optimisation
- is provided as an installable package
- may contain other artifacts, such as documentation and release notes
- may take longer to build

While agreeing that developers and users may have different requirements for the software, I consider that the phrase 'Debug build' is a poor choice.

As an example, I was recently helping to solve a problem which had been detected while running the release build of a product. The developer tried to reproduce the problem by running the debug build of the program under a debugger, but this did not fail. I suggested running the release build under a debugger (since it was the release build which demonstrated the fault) but the developer hadn't realised you could do this – they had assumed only a debug build could be debugged.

I prefer to use the descriptions 'Developer' and 'Retail' build to the more traditional 'Debug' and 'Release' build as, to my mind, this moves the spotlight onto the target audiences rather than focussing on the specific issue of debugging the program. I'll generally be using these phrases in the rest of the article.

Disadvantages with having two builds

There are several disadvantages with having two builds. Firstly, there is some duplication within the build process itself, and there is a danger that the two build streams will diverge. If you are fortunate the divergence will be caught by a compilation failure; if you are unlucky a necessary change will be made to the developer build only and the same change will not occur in the retail build. More importantly, you now have two different executables and they may not have the same bugs. The developer build usually has much more testing during product development so any problems specific to the retail build are typically only found late in the release cycle. To make this worse, these problems are build related, and so will not be found if debugging is attempted with the developer build. Some of the common causes of bugs that are visible only in a retail build are:

- Optimisation: either caused by compiler bugs, or exposing an existing bug hidden in the non-optimised build
- Use of assert or conditional code (eg trace or logging statements) with unforseen side-effects
- Memory set to fixed 'fill' values in a developer build and uninitialised in a retail build causing different behaviour

Over the years I have encountered many problems that were only present in the retail build; as well as some application bugs with different symptoms in developer and retail builds (for example, local variable layout

Roger Orr has been programming for 20 years, most recently in C++ and Java for various investment banks in Canary Wharf. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

you don't need to ship all the resultant information to your customers if you wish to preserve your company's intellectual property

re-ordering meaning pointer errors corrupted different variables). These bugs are often expensive to find and to fix because they do not occur in the standard development environment.

As I mentioned at the start of the article, my own preference where possible is to avoid having two separate builds and just have a single build. This simplifies the build process and also means the end user gets the same code that we've been testing during development. Where this is not achievable I try and bring the two builds as close to each other as possible, at least in terms of code generation. In practice nearly all tool chains provide ways to configure the system to select which characteristics are part of which build.

A 'release' build can be debugged

Even where a having two separate builds make good sense, there is often no technical reason why a retail ('release') build of the program cannot be run under a debugger. The main problems using a debugger with the usual default retail build configuration are:

1. the order of execution may not match the source code (retail builds are usually optimised)
2. there are no names for some (or all) functions (symbols are usually omitted in retail builds)
3. variables may be missing or appear to have the wrong values (a combination of the above reasons)

These problems may also affect dependent components – for example in the Microsoft world there are different C runtime libraries for their 'Debug' and 'Release' builds and there is much more symbolic information in the debug library than the release one.

If you have the source for a dependent component you can simply change the retail build settings; in some cases you may be able to push back on the supplier of third party components to deliver builds containing more symbols or using different optimisation levels.

How fast is fast enough

The first problem to deal with is the effect of optimisation. There are three potential problems with optimisation. Firstly, it makes the resultant code harder to debug; secondly optimising the code may introduce bugs and finally running an optimiser can make the whole build process take longer. How much optimising does your program really need – and where?

Much of the code in your program may not gain value from optimising and you may decide that the benefits of the developer and retail builds being more similar are worth a slight performance degradation or a slight gain in the size of the binary. Indeed, with most tool chains you can selectively enable optimising just for the parts of the program that benefit from it.

The actual decision that you make will depend on factors such as how important performance is to users of your application and how much time is spent finding and fixing problems in the retail build.

One specific optimisation which should be considered for disabling, especially on the Intel x86 architecture, is frame pointer optimisation. On stack based machines each function has a 'frame' of memory which contains the local variables, function arguments and the return address. On entry to the function the pointer to the previous stack frame is saved, and it will be restored when the function returns.

The stack frame pointers in un-optimised code normally form a chain through the stack, allowing tools to work out the call chain for the current function and identify the function arguments. This makes many debugging tasks easier as knowing 'how you got here' is often a key component to working out the root cause of a problem.

When code is optimised the stack frames can be set up in non-standard ways – the code in the function itself knows how to unwind the frame but a general purpose tool, such as a debugger, can't work back up the call stack. Both g++ (on many architectures) and Microsoft Visual C++ allow you to turn this optimisation off.

I have measured the impact of turning this optimisation off and, in my own experience, the impact has been minimal. As always with optimisation you need to measure the impact in your own specific cases.

- for MSVC use `/Oy-`
- for g++ use `-fno-omit-frame-pointer` [WildingBehman]

Microsoft themselves seem to consider the ease of debugging outweighs the performance improvement – starting with Windows XP service pack 2 the operating system itself has been compiled with frame pointer optimisation disabled. This makes it much easier for debuggers to work back up the stack from a problem detected in a system component to the application code that, for example, passed a bad parameter to a Windows API.

Names matter

Debugging programs without symbolic information is hard as all you have are assembler mnemonics and memory addresses with no idea of their usage. As John Robbins puts it: '*If you're paid by the hour, spending forever at the assembly language level could do wonders for paying your mortgage.*' [Robbins]

Both g++ and Microsoft Visual Studio allow you to add symbolic debug information to retail builds. In both cases you don't need to ship all the resultant information to your customers if you wish to preserve your company's intellectual property. I strongly recommend that you check the retail builds of your software do provide as much symbolic information as possible.

Symbols for g++

Use `-g` debugging option(s) in combination with `/O[n]` optimising options. The resultant program will be optimised and contain debugging symbols.

The Unix model by default puts all the debugging information into the exectable program. This does not usually cause any execution time overhead since the data is not loaded from disk into memory unless a debugger is being used. It does mean that the executable may be larger; in some cases considerably larger, depending on how much debugging information was created.

it can be quite hard to ensure that the right version of the file is always kept with the corresponding executable

On many platforms the debugging information can be extracted from the executable enabling use of the debug information only at the time when debugging is required. This also provides a way to restrict access to the debugging symbolic information – simply don't ship the debug information to your customers!

An example of splitting the debug information out:

1. Link the executable with **`-ggdb -O2`** producing, for example, **`prog`**.
2. Run "**`objcopy --only-keep-debug prog prog.dbg`**" to create a file containing all the debugging info.
3. Run "**`objcopy --strip-debug prog`**" to remove the debugging info from the executable.
4. Run "**`objcopy --add-gnu-debuglink=prog.dbg prog`**" to link the debugging info with the executable.

Symbols for MSVC

Visual Studio 2005 now puts debug information into Release builds by default. For projects created with earlier versions of the tools you must

- Use **`/Zi`** at compile time (in the C/C++ 'General' tab under 'Debug Information Format')
- Use **`/DEBUG`** at link time (in the Linker 'Debugging' tab under 'Generate Debug Info')
- Additionally, to reduce the exectable size, set the linker **`/OPT:REF`** and **`/OPT:ICF`** options (in the 'Optimization' tab)

Under Visual Studio the debugging symbols are stored in the PDB file with a link record in the binary file. Note that the debuggers verify that the PDB file was created by exactly the same linker execution that produced the executable file. Some debuggers do allow you to use mismatched files, but when this is possible the symbols in the PDB file may not longer have any connection with the binary addresses in the exectable program, so make sure you always keep the two files together.

Just as in the g++ case, you have the option on whether or not you ship the symbolic information to your customers; simply miss out the PDB file. There are also ways to provide PDB files with less information for public consumption and retain the full symbol files for internal use. See the program pdbcopy.exe in Microsoft Debugging Tools for Windows [MSDebug] which allows you to strip private symbols from your PDB files. Microsoft use this technique themselves for the symbols they make publicly available.

Microsoft symbol servers

There are a couple of main problems with the debugging symbol files. Firstly, it can be quite hard to ensure that the right version of the file is always kept with the corresponding executable. Secondly the files are quite large – often larger than the executable binaries – but only required when someone is actually debugging the application.

Microsoft have addressed these problems in their recent debuggers with the result that you need never be without the right symbols at the right time. The secret weapon is the symbol server engine (`symsrv.dll`) which is shipped with the Visual Studio Debugger and Windbg. This engine is able to locate the right version of the symbol file for the executable being debugged, either from a subdirectory on the hard disk or a networked drive, or using http across the network (either Intranet or Internet).

Microsoft have been providing symbols for all their retail releases of Windows and other of their products for some time now, and setting up your machine to access this information can greatly improve debugging on the Windows platform.

The engine uses the environment variable **`_NT_SYMBOL_PATH`**. This environment variable can contain multiple paths (semicolon delimited) and any path can be marked for the symbol server engine by using the syntax **`SRV*`*`cache location[*server]`***.

For example, setting the value to **`SRV*C:\Symbols*http://msdl.microsoft.com/download/symbols`** tells the debugger to look for symbol files in the cache directory of **`C:\Symbols`** and, if not found there, to look on the Microsoft Web site and download (and cache) any matching debug files.

The symbol server makes sure the right PDB is always used for the executable file by using subdirectories in the local cache and using information put in the binary by the linker to access the correct subdirectory for the executable.

So, for example, my cache directory contains several different versions of `kernel32.pdb` reflecting different versions of Windows and various hot fixes which have been applied (Figure 1).

```
Directory of U:\Symbols\kernel32.pdb
14/06/2007  20:03    <DIR>  .
14/06/2007  20:03    <DIR>  ..
02/07/2006  14:42    <DIR>  3E8016FF2
25/09/2006  22:37    <DIR>  44C5EB742
02/07/2006  14:42    <DIR>  75CFE96517E5450DA600C870E95399FF2
14/06/2007  20:03    <DIR>  7FD4C98964054C24B2C472948D829DF52
13/06/2007  01:15    <DIR>  DAE455BF1E4B4E249CA44790CD7673182
```

Figure 1

automatically add the symbols from your own application builds to a symbol store so that people debugging the program have access to the right symbols

Using the internal timestamp of the DLL automatically makes sure the right symbols are always used with no need for input from the programmer during debugging.

The downside with this approach is that the symbol server engine will look on the Microsoft website for all symbol files, even for third party DLLs. This can significantly slow down starting the debugger. My own technique is to do most debugging with `_NT_SYMBOL_PATH` containing the directory of the symbol cache but not the Microsoft website: `_NT_SYMBOL_PATH=SRV*U:\Symbols`. If I find symbols are missing for a Microsoft DLL or EXE then I attach a debugger with the full symbol path to force a download of the relevant symbols.

Adding your own symbols to the symbol store

It can make a lot of sense to automatically add the symbols from your own application builds to a symbol store so that people debugging the program have access to the right symbols. This also enables easier debugging of mini-dumps from customers since the debugger can automatically find and load the right symbols for the actual versions of the program being run at the time of the crash. There are several ways to do this, with varying levels of complexity, and I refer anyone interested in this to the Microsoft Debugging Tools documentation for a fuller explanation than this article can provide.

The simplest way is to add your own builds into the symbol store used for the files downloaded from Microsoft. The symstore program can be used to add files to the symbol store. For example, to add all the binary and symbol files from version 1.2 of 'my product' (Figure 2).

This step can be added to the automated build for retail versions of your product to ensure the binaries are collected. Depending upon disk space you might need to purge old versions (or pulled releases) from the symbol server, but compare the costs of disk space to programmer time before deleting any files.

Conclusion

The common paradigm of having 'Debug' and 'Release' builds has some utility, reflecting the different needs of developing or testing code and running it 'for real'. I prefer to name the two builds 'Developer' and 'Retail' builds to express their intent more clearly.

However, there are downsides to having two different builds and it is worth making an informed choice about whether the benefits outweigh the costs.

Should you choose to retain two builds, the retail build is likely to need some debugging and it is well worth spending some time up-front to make sure that this task will be as easy as possible. An important part of this is to ensure that the debugger has maximal access to any available symbolic information. ■

Acknowledgements

Thanks to the Overload review team for the various improvements they suggested for this article.

References

[MSDebug] http://www.microsoft.com/whdc/devtools/debugging/default.mspx

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

[WildingBehman] *Self-Service Linux*, Mark Wilding and Dan Behman, Prentice Hall

```
C:symstore add /r /s C:\symbols /t MyProduct /v 1.2 /f C:\MyProduct\Build
Finding ID...  0000000321

SYMSTORE: Number of files stored = 107
SYMSTORE: Number of errors = 0
SYMSTORE: Number of files ignored = 576
```

Figure 2

auto_value: Transfer Semantics for Value Types

"Copy On Write" (COW) sounds like an ideal idiom for avoiding expensive copies. But care must be taken to avoid producing a "mad cow".

Last time we took a look at the various flavours of smart pointer and I suggested that **auto_ptr** could be simplified by separating the lifetime control and ownership transfer responsibilities into two classes, **scoped_ptr** and **auto_ptr** respectively.

I closed with a brief mention of the copy-on-write, or COW for short, optimisation for strings and the fact that it relies upon one of the smart pointers, **shared_ptr**.

This time, we'll take a detailed look at **string**.

string

Before we describe how COW works, let's take a look at a naïve implementation of a string class (Listing 1).

A **scoped_array** (identical to **scoped_ptr**, except that it uses **delete[]** instead of **delete**) ensures that the data is deleted when a string goes out of scope.

```
class string
{
public:
  typedef char         value_type;
  typedef char *       iterator;
  typedef char const * const_iterator;
  typedef size_t       size_type;
  //...

  string();
  string(const char *s);
  string(const string &s);

  string & operator=(const string &s);
  string & operator=(const char *s);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  size_type size_;
  scoped_array<char> data_;
};
```

Listing 1

Richard Harris Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

```
string::string() : size_(0), data_(0)
{
}


string::string(const char *s) : size_(strlen(s)),
   data_(size)
{
  std::copy(s, s_size_, data_.get());
}

string::string(const string &s) : size_(s.size_),
   data_(new char[size_])
{
  std::copy(s.data_.get(), s.data_.get()+size_,
     data_.get());
}
```

Listing 2

The constructors are pretty straightforward (Listing 2).

The copy and conversion constructors allocate new strings whose lifetimes are managed by the **scoped_array** member **data_**. The constructor bodies then simply copy the strings. Nothing particularly surprising.

The assignment operators are a little more complex, though. These days the recommended way to implement assignment is with the copy and swap idiom.

This is usually expressed as:

```
T &
T::operator=(const T &t)
{
  T tmp(t);
  swap(tmp);
  return *this;
}
```

The chief advantage of this approach (besides its relative simplicity) is that it is guaranteed to leave the object in its original state if an exception is thrown during the copy operation. This is because we don't commit the change until we swap the object with the temporary copy and the call to swap is guaranteed not to throw.

In the case of **string**, the **swap** member function could be defined as:

```
void
string::swap(string &s)
{
  std::swap(size_, s.size_);
  std::swap(data_, s.data_);
}
```

Unfortunately **std::swap** will need to assign a new value to both **data_** and **s.data_** and since **scoped_array** doesn't allow us to rebind the

it seeks to eliminate unnecessary copies by deferring them until they can no longer be avoided

pointer, it doesn't provide an assignment operator or **reset** member function. If we want **scoped_array** to consider classes as scopes, we'll need to provide some mechanism to enable assignment.

Adding assignment operators to **scoped_array** would dramatically weaken its guarantee that it will only ever point to one array.

Thankfully, we can achieve what we want by adding a **swap** member function instead, defined as follows:

```
void
scoped_array::swap(scoped_array &a)
{
  std::swap(x_, a.x_);
}
```

This still weakens **scoped_array**'s guarantee, but in a way that's less appealing to use for ownership transfer than an assignment operator or **reset** member function. Well, that's my story and I'm sticking to it.

We can now redefine **string**'s **swap** member:

```
void
string::swap(string &s)
{
  std::swap(size_, s.size_);
  data_.swap(s.data_);
}
```

Now, the naïve approach is perfectly good for short strings, but those string copies in the constructors, and indirectly in the assignment operator, start to look pretty ominous in the face of very long strings.

For example, consider the sequence of events when storing the result of a function call (Listing 3).

So what exactly happens when we call **g**?

```
string
f()
{
  string s("hello, world");
  return s;
}

void
g()
{
  string s = f();
}
```

Listing 3

```
call g

call f
copy "hello, world" into s
copy s into return temporary
exit f

copy return temporary into s
exit g
```

Yikes. I count two completely unnecessary copies of the data. Now that's not such a problem for "hello, world", but will hit hard for the King James Bible, for example.

So how does copy-on-write help?

Well, it seeks to eliminate unnecessary copies by deferring them until they can no longer be avoided. This is typically when an element of the string is about to be changed, hence the name. Until that moment, a 'copy' of a string simply holds a reference to the original.

This is related to, but subtly different from, reference counting for shared ownership. Unlike shared ownership, COW allows multiple references to the underlying data of an object only so long as the external behaviour is unaffected.

With shared ownership, we expect multiple references to be able to change the observed state of the object. With COW, this must be avoided at all costs. We use reference counting merely to avoid making multiple copies of provably identical data.

For example:

```
void
f()
{
  string s1("hello, world");
  string s2(s1); //s2 can hold a reference to
                 //s1's data here
  std ::cout << s2 << std::endl;
  s2.replace(0, 5, "goodbye");  //s2 must copy
                                //s1's data here
}
```

Until we actually change the state of **s2** in the last line of **f**, the behaviour of the function is unchanged whether **s1** and **s2** share their state or not.

Let's have a look at how we might implement a string class that supports COW (Listing 4).

The point to note about this definition is that the data is stored in a **shared_array** (like **shared_ptr**, except that it uses **delete[]**) rather than as a **scoped_ptr**. With this we will share, rather than copy, data for as long as possible.

To see how this works, let's take a closer look at a few member functions.

First, the constructors (Listing 5).

the same check for **uniqueness and subsequent copy** must be present in every function that presents an opportunity to change the string

```
class string
{
public:
  typedef char          value_type;
  typedef char *        iterator;
  typedef char const *  const_iterator;
  typedef size_t        size_type;
  //...

  string();
  string(const char *s);
  string(const string &s);

  string & operator=(const string &s);
  string & operator=(const char *s);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  size_type size_;
  shared_array<char> data_;
};
```

Listing 4

The sharp-eyed amongst you will have noticed that the copy constructor is superfluous since the compiler generated version would have done exactly the same thing.

```
string::string() : size_(0), data_(0)
{
}

string::string(const char *s) : size_( strlen(s)),
                                data_(new
char[size_])
{
  std::copy(s, s+size_, data_.get());
}

string::string(const string &s) : size_(s.size_),
                                  data_(s.data_)
{
}
```

Listing 5

The important point though is that when we construct a **string** with a C-style point to character array, the data is copied, whereas when we copy construct a **string**, the data is shared.

Now let's have a look at the two flavours of **begin** to see how we make sure that we don't accidentally change a **string**'s value through this shared data:

```
string::const_iterator
string::begin() const
{
  return data_.get();
}
```

Well, the first version is pretty simple. Since we're returning a **const_iterator** (defined as a pointer to **const T**), it's not possible to change the contents of the string so we can simply return the pointer to the start of the character data.

Granted, **const_cast** could throw a spanner in the works, but I doubt anyone would be too upset if we simply declared casting between iterator types undefined behaviour and ignored the problem.

Clearly, it's the non-**const** version of the function that is of interest:

```
string::iterator
string::begin()
{
  if(data_.get() && !data_.unique())
  {
    char *s = data_.get();
    data_.reset(new char[size_]);
    std::copy(s, s+size_, data_.get());
  }

  return data_.get();
}
```

And here we see the mechanism at work. If more than one **string** is referring to the data, we copy it before we allow a means to change it to escape. The same check for uniqueness and subsequent copy must be present in every function that presents an opportunity to change the string.

In the following example:

```
void
f()
{
  const string s1("hello, world");
  string s2(s1);
  string::iterator i = s2.begin();
  *i = 'y';
}
```

for multi-threaded programs, the problem is that sharing and releasing the string data are not atomic operations

we have the following sequence of events:

```
construct s1
copy "hello, world"

construct s2
share "hello, world"

s2.begin
fail uniqueness check
copy data
return iterator

assign 'y' to start of s2
```

Given that we have already established the sharpness of your eyes, I have no doubt that you have raced ahead of me and spotted that this code isn't remotely fit for purpose.

To hammer home why, consider the following example:

```
void
f()
{
  string s2("hello, world");
  string::iterator i = s2.begin();
  const string s1(s2);
  *i = 'y'; //oops, s1 has changed too
}
```

Let's take a look at the sequence of events this time:

```
construct s2
copy "hello, world"

s2.begin
pass uniqueness check
return iterator

construct s1
share "hello, world"

assign 'y' to start of s1 and s2
```

It seems there was a potential alias that we overlooked, the iterator itself.

This is harder than I expected. Perhaps I should be more forgiving of my unnamed library vendor.

Worse still, even if we correctly identify and protect all possible aliases, we still have made a rather sweeping assumption. Namely, that `string`s will only be referred to by a single thread.

In a multi-threaded program, it is quite possible that a COW string's data could be manipulated by more than one thread at the same time. To illustrate the problems that this can lead to, consider what happens when two copies are made of a string.

```
void
f(const string &s)
{
  string s1(s);
}

void
g(const string &s)
{
  string g1(s);
}
```

In a single threaded program, `f` and `g` must be called sequentially:

```
string s("hello, world");
f(s);
g(s);
```

leading to the following sequence of events:

```
construct s
copy "hello, world"

call f
share "hello, world"
release "hello, world"
exit f

call g
share "hello, world"
release "hello, world"
exit g
```

and everything goes smoothly.

For multi-threaded programs, the problem is that sharing and releasing the string data are not atomic operations. Specifically, they must read the reference count, manipulate it and finally write it. Unfortunately a thread may be interrupted during these steps.

For example, if we were to call `f` and `g` on separate threads:

```
string s("hello, world");
run_threaded(f, s);
run_threaded(g, s);
```

we might observe the following sequence of events:

our checks end up taking longer than making a copy of the string, rendering the entire approach rather pointless

```
construct s
copy "hello, world"

call f
call g

f: read reference count       (count == 1)
f: increment reference count  (count == 2)
g: read reference count       (count == 1)
f: write reference count      (count == 2)
g: increment reference count  (count == 2)
g: write reference count      (count == 2)
```

oops

Because **g** read the reference count before **f** had committed its change, we've lost the record of **f**'s interest in the string. This is certainly going to lead to interesting behaviour at some point in the program.

To protect ourselves from this eventuality, we need to ensure that only one thread can read or manipulate the reference count at any given time.

Fortunately there's a specific threading tool to do this, the mutex (or mutual exclusion) lock. Unfortunately it's not free. And since we need to lock each uniqueness check we'll need a lot of locks. So many that it's perfectly possible that our checks end up taking longer than making a copy of the string, rendering the entire approach rather pointless.

I've heard of at least one string implementation that sought to reduce the number of mutex locks by having a single mutex for every string in the program. This does cut down on the cost of creating locks, but has the unfortunate side effect that every access of every string is synchronised, rather defeating the point of multi-threaded string manipulation.

So is it worth it?

For my part, the chief justification for using COW is that I hate temporaries. Or, more accurately, I hate the expense of copying them left, right and centre. Winds me right up, it does.

Recall that with our naïve string, storing the result of a function call involved two unnecessary copies of the data.

Hold on a sec.

Did I say two unnecessary copies?

Doesn't the C++ language specifically allow compilers to avoid making unnecessary copies?

There I go with my little lies again. It's been quite a while since compilers would create any temporaries at all in this situation. C++ specifically allows such temporaries to be side-stepped ('elided', in the terminology of the standard) and the result of the call to **f** to be constructed directly into **s**.

There are two situations where a C++ compiler is legally allowed to avoid copying an object.

Firstly when a function return expression is the name of a local object of the same type as the return value, the object can be constructed directly into the return value rather than copied. For example:

```
string
f()
{
  string s;
  s = "hello, world";
  return s;
}
```

In this case, rather than constructing **s**, manipulating it and copying it into the return value a C++ compiler can, by treating **s** as an alias for the return value, simply construct the return value and manipulate it directly, saving a copy.

Secondly when a temporary that has not been bound to a reference would be copied to an object of the same type, the temporary can be constructed directly into the object. For example:

```
string t = f();
```

In this case, the temporary return value of **f** can be treated as an alias of **t**, saving a copy.

Note that the two optimisations can be applied to the same statement. In the above example, this means that the string **s** in the function **f** can be treated as an alias for the string **t**, effectively eliminating two copies.

So are there any situations where unnecessary copies still exist?

Well, firstly there's the case when strings are passed by value, but are not consequently changed. But **const** references capture this behaviour perfectly, so this doesn't seem to be particularly compelling.

Secondly there's the case when a function has multiple points of return. For example:

```
string
f(bool b)
{
  string s, t;
  s = "hello, world";
  t = "goodbye, world";

  if(b) return s;
  return t;
}
```

In such cases it can be difficult for the compiler to predict which of the return expressions should be treated as an alias for the return value. In this case, should the compiler pick **s** or **t**?

Of course, a simple restructuring of the code would make things easier for our beleaguered compiler:

```
string
f(bool b)
{
  string s;
  if(b) s = "hello, world";
  else  s = "goodbye, world";
  return s;
}
```

But there will inevitably be situations where such restructuring is difficult, or at least unwieldy.

Finally there's the case when a temporary is assigned to a string. In this case COW may save us a copy. (Listing 6.)

The reason why COW won't always save a copy in this case is a little subtle.

As I mentioned before, the recommended way to implement assignment is with the **copy** and **swap** idiom:

```
T &
T::operator=(const T &t)
{
  T tmp(t);
  swap(tmp);
  return *this;
}
```

In this form, we will definitely pay for a copy during assignment of a naïve **string**. This is because the compiler binds the temporary to a **const** reference, rather than to a newly constructed object so it can't be elided. On the following line we copy construct the temporary, but by then it's too late, since the function can't know if the reference is to a temporary or a named variable.

Fortunately, we can rewrite the code to take advantage of copy elision:

```
T &
T::operator=(T t)
{
  swap(t);
  return *this;
}
```

Now the temporary is passed directly to the copy constructor of the named argument, and is therefore a candidate for the copy elision rule. The result of a function call can be constructed directly into **t** which is then swapped with the object.

So that's that for COW then, isn't it?

Well, COW can also save us a copy or two when we might need to change a string, but aren't certain.

For example, consider a function to strip a trailing newline from a **string** (Listing 7).

```
string
f()
{
  string s("hello, world");
  return s;
}

string
g()
{
  string s("goodbye, world");
  return s;
}

void
h()
{
  string s = f();
  //...
  s = g(); //COW may save us a copy here
}
```

Listing 6

```
string
f(string s)
{
  if(!s.empty() && *s.rbegin()=='\n')
  {
    return s.substr(0, s.size()-1);
  }
  else
  {
    return s;
  }
}

void
g()
{
  string s("Hello, world\n");
  //...
  std::cout << f(s) << std::endl;
}

void
h()
{
  string s("Hello, world");
  //...
  std::cout << f(s) << std::endl;
}
```

Listing 7

we'd have to let the newline stripping logic **leak out** into the calling function

```
void
f(const string &s)
{
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) <<
std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}
```

Listing 8

In the function **g**, **string** makes a copy since the original **string** has a trailing newline. In **h**, however, no change is required so no copy will be made.

```
void
g()
{
  string s("Hello, world\n");
  //...
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) <<
std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}

void
h()
{
  string s("Hello, world");
  //...
  if(!s.empty() && *s.rbegin()=='\n')
  {
    std::cout << s.substr(0, s.size()-1) <<
std::endl;
  }
  else
  {
    std::cout << s << std::endl;
  }
}
```

Listing 9

```
void
f(const string &s)
{
  if(!s.empty())
  {
    string::const_iterator first = s.begin();
    string::const_iterator last  = s.end();

    --last;
    while(first!=last) std::cout << *first++;
    if(*first!='\n')   std::cout << *first;
  }
}
```

Listing 10

We can get the same benefit, however, if we rewrite the code so that a copy is only made if we need one. For example, Listing 8 or Listing 9.

OK, I'll admit it, this isn't really a very compelling argument. We'd either need a different version of the function for every operation we want to perform on the resulting string or we'd have to let the newline stripping logic leak out into the calling function, neither of which are particularly attractive prospects.

Well, that and the fact that a more sophisticated COW string wouldn't make a copy if you were just creating a sub-string. Adding offset and length members to **string** would allow sub strings to share a reference into the original string, delaying the copy until we do something really destructive like change some characters.

Nevertheless, I contend that this aspect of COW does not confer that big an advantage. Consider Listing 10.

This still has the unfortunate property that we'd need one function for each operation, but gains the significant advantage of making no copies whatsoever. Furthermore, with a few minor changes, we have something that looks suspiciously like a function from **<algorithm>** (Listing 11).

This being a generic algorithm to perform an operation, in our case printing, on every element in the iterator range except the last if it is equal to **t**.

```
template<class BidIt, class T, class UnOp>
void
f(BidIt first, BidIt last, T t, UnOp op = UnOp())
{
  if(first!=last)
  {
    --last;
    while(first!=last) op(*first++);
    if(*first!=t)      op(*first);
  }
}
```

Listing 11

if you really care about the efficiency of your string processing, I very much doubt that you would rely on delayed copy optimisation

Personally, I tend to view the STL less as a library than as a way of life, or at least I did until my girlfriend threatened to leave me if I didn't start washing once in a while. Now I guess I see it more as a way of programming.

If the STL doesn't have the algorithm or data structure I want I implement it myself and add it to my own extensions library. This can be a lot of work at the outset, but starts paying dividends fairly quickly.

I doubt I'd consider this a candidate for my extensions library, but my point is that if you really care about the efficiency of your string processing, I very much doubt that you would rely on delayed copy optimisation. A much better approach is to think very carefully about the algorithms you need to perform and to implement them in an efficient manner.

So, given my assertion that we can write our code in such a way that copy-on-write optimisation is unnecessary and that it has synchronisation problems to boot, is this kind of approach really worth further consideration?

Well, the growing opinion is no. Some string implementations (notably STLport) are turning to alternative optimisations such as the short string optimisation and expression templates.

The short string optimisation involves adding a small array member to the string class. (Listing 12, for example.)

```
class string
{
public:
  typedef char         value_type;
  typedef char *       iterator;
  typedef char const * const_iterator;
  //...

  string();
  string(const char *s);
  string(const string &s);

  string & operator=(const string &s);
  string & operator=(const char *s);

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...

private:
  char data_[16];
  char *begin_;
  char *end_;
};
```

Listing 12

When the string is less than 16 characters long the array **data_** can be used to store it in its entirety, eliminating the need for a relatively expensive allocation when copying. Longer strings must be allocated from the free store as usual. The **begin_** and **end_** pointers are used both to manage the extent of the string and determine whether the internal array or the free store have been used to store it, enabling the destructor to ensure that the memory is correctly released when the string is destroyed.

Note that this optimisation does not reduce the number of characters that are copied when strings are copied. Instead it relies upon the speed of allocating and copying memory on the stack rather than the free store.

Expression templates are altogether more complicated beasts and their precise mechanics are beyond the scope of this article. Put simply they work by deferring string manipulation operations until the result is actually assigned to something. For example, in the code snippet:

```
string s1 = "Hello";
string s2 = "world";
string s3 = s1 + ", " + s2;
```

the strings **s1**, **", "** and **s2** are not actually concatenated until **s3**'s constructor requires the result. At this point a triple concatenation is performed, rather than the two double concatenations that a simple string implementation would perform. This saves both the creation of a temporary sub string and copying it into the final string.

In fact, expression templates are a very powerful technique that can be used for far more sophisticated optimisations than simply eliminating copies.

Despite this, I'm not yet willing to dismiss COW like optimisations out of hand, although you'll have to wait until next time before I explain why. ■

Acknowledgements

With thanks to Kevlin Henney for his review of this article and Astrid Osborn, Keith Garbutt and Niclas Sandstrom for proof reading it.

Guidelines for Contributors

Thinking of writing for us? Follow these guidelines to help smooth the way.

These guidelines provide general instructions on the submission of articles for publication. For more detailed information, please contact the editor of the relevant publication (cvu@accu.org or overload@accu.org).

For examples of the elements described, please see recent issues of the journals.

With your article, you need to send:

- A short personal profile (see 'Profile')
- Any images used in your article as separate files (see 'Illustrations')
- An introductory line or sentence (see 'Structure')

Format

Articles can be accepted as Word or Open Office documents; alternatively, save your file as RTF. If your article is in any other format, please check with the production team that it can be opened and the text extracted.

If you are using a text editor, devise and explain a convention for your document. For example, 'Heading 1 in block capitals, heading 2 underlined, code snippets in text surrounded by <<c>>'. You can use a numbered system for headings to help with this – the numbers will be removed from the finished article. Clearly separate any such instructions from the body of the document.

Text for sidebars or panels can either be inserted in the approximate position in the text or provided at the end if its location is unimportant. Clearly mark the beginning and end of the text forming the sidebar.

Please do not apply complex formatting to your articles – this will almost certainly be changed during the typesetting process to comply with the journal standards. The formatting you apply is, however, used as a guide by the production editor when determining how to format particular elements.

A fixed width font is used to display code fragments and filenames (bold for code and non-bold for comments and filenames) – if possible, use a suitable font in your article to indicate this (Courier, for example).

The formatting of listings will almost certainly be changed due to space constraints. Unless there is good reason to do otherwise, code is placed in a single column, giving a maximum number of characters per line of 48. If the formatting of your code is important to you, you are welcome to ensure your listings fit within this number of characters and every attempt will be made to retain your personal style, although this cannot be guaranteed. If the formatting of the code is particularly relevant to a point you are making (for example, contrasting two different styles of writing), please say.

A Word template is available containing a Code paragraph style set to the correct width.

Profile

A short profile of the author is required for all articles. This should be brief – approximately 50 words – and may need to be shortened if space is short.

Please ensure your name is as you would wish it to appear in the journal and include an email address or another means of contact (for example, via a website).

For C Vu only, a photograph of the author (head and shoulders) is also required. This should be high-contrast and at a minimum resolution of 200 dpi (ideally 300 dpi).

Structure

Try to keep titles short and relevant – they should ideally fit on a single line and definitely require no more than two lines. A short strapline (C Vu) or introductory sentence (Overload) is required – if these are not provided by the author, they will be created as part of the editorial process.

Both publications have provision for three levels of heading within the body of the article, although the third should be used rarely. Headings are not numbered, so please do not reference other sections by heading number in your article, even if you use them to indicate levels.

Illustrations

You must supply images as separate files, as well as placing them in your article to show approximate location. The journals are printed in black and white, so please make sure that your images do not rely on colour alone to differentiate between the components. Also check that any lettering can be seen when the image is converted to greyscale and it has been scaled to fit on a page. (Note: Colour images are still useful, as a PDF version is produced in colour).

- Supply images in common formats such as JPG, TIFF, or PNG.
- Photographs and similar images must be a minimum of 200 dpi, ideally 300 dpi, at the size to be printed.
- Save diagrams and line drawings in vector format if possilbe. Do not attempt to convert raster images to vector format.
- Screenshots should be saved in a non-lossy format. TIFF and PNG have been used successfully. GIF is also a suitable format, but check that the colour depth supported by your application has not adversely affected the image.

References and Notes

CVu and Overload treat references and footnotes differently. Please ensure that references are as complete as possible.

- CVU: References and notes are marked by placing numbers in brackets in the text; for example: [1]. References and notes are both numbered in the same sequence and the corresponding information provided in a Notes and References section at the end of the article.
- Overload: Notes are treated as footnotes - number these and supply the text either as footnotes or at the end of the article. References are marked in the text using the author's name and final two digits of the year of publication (for example, [Griffiths06]). The full reference is placed at the end of the article. If you cannot specify an author or year, choose a suitable word. For example, if you are referencing the Boost libraries, use [Boost] as the reference marker. ■