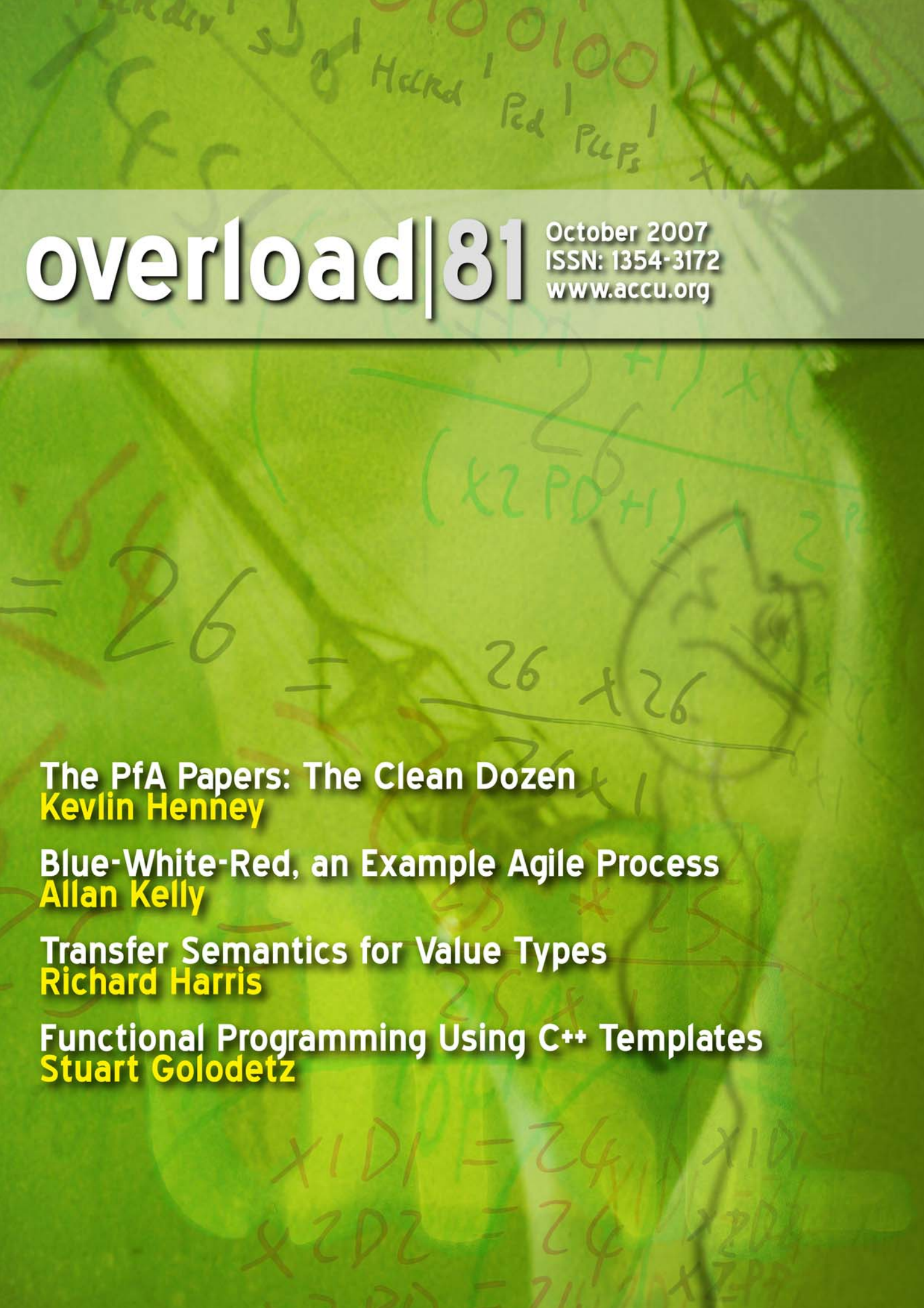
overload|81
October 2007
ISSN: 1354-3172
www.accu.org



The PfA Papers: The Clean Dozen
Kevlin Henney

Blue-White-Red, an Example Agile Process
Allan Kelly

Transfer Semantics for Value Types
Richard Harris

Functional Programming Using C++ Templates
Stuart Golodetz

OVERLOAD 81

October 2007

ISSN 1354-3172

Guest Editor

Ric Parkin
ric.parkin@gmail.com

Editor

Alan Griffiths
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Farnsworth
simon@farnz.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 82 should be submitted to the editor by 1st November 2007 and for Overload 83 by 1st January 2008.

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

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

While the Cat's Away...

…instead of spending your time playing, why not learn to be multi-lingual in a multi-cultural world?

Hello and welcome to Overload #81.

In a change to our scheduled programming, I've taken over editing this issue while Alan's been away. And thanks to the sterling efforts and helpful advice from everyone involved it has all been remarkably stress-free for me (well apart from the moment when I realised I'd have to write an editorial – I thought I'd stopped doing essays years ago!)

But it's all been worth it and we've another great issue for you with highlights including the next installment in the thrilling soap opera that is 'Paramaterise from Above', and Allan Kelly showing us some card tricks.

Raising standards

In Overload #80, the editorial talked about some odd things happening in the Standardization organisations. Since then there have been more reports of irregularities in the OOXML process [Groklaw], with several national groups raising objections, both to the standard and the way the votes have been conducted.

I won't comment on what's going on – that's a long conversation over a few beers I suspect – but one thing struck me as very interesting: the Internet Society Netherlands suggested [ISN]:

> ISOC.nl recommends that the ISO procedures – and more specific the Fast Track procedure – be adapted significantly to better deal with controversial standards like DIS 29500/Office Open XML in order for ISO to maintain relevant. This includes demanding two interoperable and independent full implementations prior to accepting a submission for a Fast Track procedure.

This reminded me of the heroic failure that is 'export' in C++: an interesting idea that was a theory without practice when it was standardized. In the subsequent years, I only know of a single implementation [EDG] that has actually appeared after a huge effort, and the reported benefits were much less than originally hoped for. Other vendors have reportedly decided never to even try to support it, reasoning that the cost/benefits just don't stack up. If an attempt to implement it had been tried before it was standardized, I think it would have been clear quite quickly that it wasn't going to be worth the effort and it would have been abandoned.

Fortunately this lesson appears to have been heeded, as the more interesting parts of the next C++ standard (usually dubbed C++0x) are being tested out in real compilers and libraries to prove their worth and smooth out the resulting implementations before they get into the standard. After all, a lot of standardization is *meant* to be about formalizing existing practice so that alternative vendors can produce a new implementations and existing code will be able to work with any implementation. In this sense, it's the interface between the compiler vendor and the user that gives guarantees about what should happen (and makes clear what is undefined or implementation defined, so you know when you're straying into such grey areas). Looked at like this it makes sense that something like `export` didn't work – we all know how bad an interface can be if it has been written in isolation without it actually being used.

One of the new features that does look very well thought through makes an appearance in the final part of Richard Harris's `auto_value` series. I feel confident that future articles will start to cover more of the new language features as time goes on.

Multi-lingual

How many languages do you know? I don't mean the usual embarrassing haul of English with a smattering of Restaurant French half remembered from school, but how many programming languages? And how many *types* of languages? After all, if you know C#, VB.net isn't a big leap, but a dynamic language such as Python does things quite differently, and a functional language such as Lisp requires a completely new way of thinking.

So what if you follow the advice 'Learn at least a new language every year' [Pragmatic]? If you know multiple types of languages, you'll have a much richer set of tools and ways of thinking about a problem, so if something is hard in one language but is very natural in another, you're more likely to be able to choose the right tool for the job or gain new insights by bringing the ideas of one language into the domain of another.

A good example of such cross-fertilization is Lambda functions – originally found in functional languages, partially implemented in boost.lambda [Boost], and now proposed for C++0x [WG21]. Another is the whole idea of functional programming itself, whose parallels with some C++ features Stuart Golodetz investigates.

Nuts and bolts

One of the main pieces of IT news in recent weeks (no, NOT the iPhone) has to be the EC's anti-competitive ruling on Microsoft with a record fine of €497M. While that's small change compared to their income and can easily be absorbed, the wider effects will only become clear over time (assuming an appeal isn't launched). While the rulings on the bundling of

Ric Parkin has been programming professionally for nearly 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him and is now organising the ACCU Cambridge local meetings. He can be contacted at ric.parkin@gmail.com.

certain applications such as Media Player is fascinating stuff (especially as I'm someone who recently bought a new PC and was frustrated to find I couldn't play a DVD so gave up and downloaded VLC instead), I was more interested in the rulings on publishing server interoperability protocols. In and of themselves, they're not very interesting, but the principle is – how much should a company publish product APIs that allow a competitor's products to talk to it?

I don't think there's an easy business answer to that one – it depends on many things such as the technology, market size and the company dominance. For example, a small company that's invented a new technology may wish to open up its protocols and let other people make compatible products in the hope that it will drive adoption faster than they could on their own, at the cost of having a smaller share of a much bigger market. Different situations will result in a different answer.

But why not publish all protocols and APIs? What would happen? Think about it as a thought experiment for a moment and you'll spot a lot of the arguments on both sides. Revealing too much may lead to a company's hard work and clever ideas being copied, and they get no benefit so won't bother again – the hope of reward drives a lot of innovation. But conversely the ability to 'mix and match' parts provides a market for someone who can make a better and cheaper widget, as long as it can work with the rest of the system, so having open protocols encourages a richer eco-system of suppliers, also driving innovation. Obviously there's a balance somewhere – just where is what this ruling was trying to establish in this particular case, and Standards bodies have a similar intent to define the points of interoperability such that you *do* have a choice of what goes on either side, just the like the size standards of nuts and bolts [ISO].

"Learn as if you were to live forever"

I've been interviewing a lot recently due to expanding our team, and have got to wondering what exactly to look for in a software developer. We tend to follow Joel Spolsky's advice in *The Guerrilla Guide To Interviewing* [Spolsky] where he recommends trying to find 'Smart people who get things done'. Now that's fine as far as it goes, but I'd go one further: 'Smart people who *get things done and want to improve*'.

I think this is vital because this industry doesn't stand still – there are always new technologies coming out, new competitors with a rival product, and more and more things become possible as power and storage improve. So you want people who'll be able to adapt and do new things – personally I don't care too much if a candidate doesn't know that much about the ins and outs of a particular area unless they're being hired as an expert for that niche, but I think it's vital that they'll learn what they need very fast and not stay stuck in a rut trying to solve every problem with the same tools and ideas.

It reminds me of an anecdote I heard years ago about someone trying to find what made someone into a Guru. They interviewed all the people regarded as the Ones Who Know to find out what was their super-hero ability – the things that made them so respected and productive – with the idea that if they trained others this core skill set they'd have lots of experts really quickly. When the results came in, they found that there was only one common skill – they used the help system to find the answers they didn't know and they used it a lot. The only thing that made them better than everyone else was the ability and desire to know more.

So what sort of things would show that someone wants to learn and are one of the better programmers? Well, if they did things like going on training courses or being mentored, going to conferences, reading programming mailing lists and magazines, meeting with like-minded people to exchange ideas.... in fact a lot like what the ACCU provides. ■

References

[Boost] www.boost.org/libs/lambda/

[EDG] Edison Design Group's C++ front end http://www.edg.com/ used by compiler vendors such as Comeau http://www.comeaucomputing.com/

[Groklaw] http://www.groklaw.net/article.php?story=20070902123701843 http://www.groklaw.net/article.php?story=2007081708383138

[ISN] http://isoc.nl/michiel/nodecisiononOOXML.htm

[ISO] http://www.engineersedge.com/iso_hardware_menu.shtml

[Pragmatic] http://www.pragmaticprogrammer.com/loty/

[Spolsky] http://www.joelonsoftware.com/articles/GuerrillaInterviewing3.html

[WG21] www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n1958.pdf

The PfA Papers: The Clean Dozen

Patterns are social animals and are often found in the company of others. This issue, Kevlin Henney looks at where 'Parameterise from Above' has been hanging out.

In this second instalment of 'The PfA Papers' we look at how the PARAMETERIZE FROM ABOVE (PFA) pattern came of age. It got a name for itself and joined a touring company of recommendations, the *Programmer's Dozen*.

Last time [Henney2007] I described the genesis of the (to date undocumented) PARAMETERIZE FROM ABOVE pattern in a review of a system for a client of mine. It was described originally under the heading of 'Parameterisation from the Top'. The theme of the original write-up was to (1) make globals parameters rather than globals and (2) move them to the top layer of the system, from where they would be passed down as parameters. This upward movement is the key and applies to many kinds of objects, not just simple configuration constants.

Relative configuration

The concern most commonly addressed by PFA is that of configuration, a common motivation (or excuse) for globals of different types: global constants scattered arbitrarily across different headers and classes or coincidentally grouped into a single uncohesive header; global objects representing raw or encapsulated handles to external resources, such as registries and configuration files; free functions offering sometimes stateful APIs to these external resources; SINGLETON objects that offer apparent absolution of the guilt of globals through the blessing of a recognised pattern. Free functions in a namespace or `static` methods in a class may have scope etiquette, but they are globally accessible nonetheless.

In discussions I had at the time with Hubert Matthews he suggested the name 'Configuration from Above'. And I am pretty sure that it was Hubert rather than me who suggested the more relative – '... from Above' – rather than absolute – '... from the Top' – naming of the practice. This name also plays to the recursive, recurrent and typically scale-free nature of software structure [Potanin+2002]. The practice is not simply about hoisting the configuration decisions all the way up to the top of the system, the idea applies consistently to any calling, usage or client-supplier relationship we find, whether a layer-to-layer, a package-to-package, a class-to-class or a function-to-function relationship. Importantly, even when there is no obvious 'top' to speak of, the recommendation still applies.

The discussions we had also made clear to me the styles of usage and range of applicability of this pattern, clarifying in my mind that although configuration was one of the most common motivators, it was not the only one. The parameterization applies to many cases where we pull out some common parameter – whether a runtime variable or a compile-time type – and have the user (or its user) responsible for making the policy decision. Hence my preference for sticking with a name that focused on parameterization rather than just configuration.

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.

Programmer's dozen

The shift in name from 'Parameterization from Above' to 'Parameterize from Above' was motivated by considering PFA to be an active recommendation rather than simply a passive description of structure. This took place – and the pattern took its place – in the context of a set of thirteen recommendations I named the *Programmer's Dozen*.

Like PFA, the *Programmer's Dozen* grew out of code and design consultancy on a client's C++ code base. In looking for a way to organise and present specific recommendations, I found twelve headings that worked well, each one phrased positively as a guideline. I published these guidelines in two of my *C++ Workshop* columns in *Application Development Advisor* [Henney2002a] [Henney2002b].

However, even before these two articles had been published, during the many months between article submission and the echo of print, I had consolidated, reorganized, generalized and augmented the recommendations. Some of the recommendations were merged because they could be considered two different takes on the same guideline. Some new ones were introduced, which is when PARAMETERIZE FROM ABOVE made its appearance. The recommendations were also no longer presented as C++-specific recommendations: I had found them useful in Java-related work, as well as elsewhere. And there were no longer twelve of them: there were thirteen. The creative tension between my lingering fondness for the numerical convenience of the original dozen and the obvious inequality with the number thirteen ultimately inspired the name *Programmer's Dozen*.

I first presented the *Programmer's Dozen* in a talk at the JAOO conference in Aarhus in 2002 [Henney2002c]. I have had a long-standing interest in the idea of 'less code, more software' and 'decremental development', and I found that the thirteen recommendations tended to enforce this. This was one reason they were presented under the heading of 'Minimalism: A Practical Guide to Writing Less Code'. That, and the fact that I submitted the talk abstract before realising that the dozen would fit the bill! It was a short talk, so each recommendation took only a slide. Here is the presentation of PFA:

- Global packages, Singletons and bundles of constants should be rationalised or eliminated. They introduce coincidental coupling, so code is less adaptable and harder to test.
- Use arguments and interfaces to invert dependencies.

There is, of course, (a lot) more to say on it than just this, but all the ingredients are there – including an explicit mention of SINGLETON. Later presentations also tended to use the phrase 'parameterize with parameters', which helps make both the motivation and the style of PFA clear.

Although the names and ordering of the *Programmer's Dozen* changed a couple of times from 2002 to 2004, the actual set of recommendations has

been stable since its inception. Here's the full dozen as they currently stand:

0. Prefer Code to Comments
1. Follow Consistent Form
2. Employ the Contract Metaphor
3. Express Independent Ideas Independently
4. Encapsulate
5. Parameterize from Above
6. Restrict Mutability of State
7. Favour Symmetry over Asymmetry
8. Sharpen Fuzzy Logic
9. Go with the Flow
10. Let Code Decide
11. Omit Needless Code
12. Unify Duplicate Code

They fall into four categories: (0-2) code meaning; (3-7) code dependencies; (8-10) code execution; (11-12) code consolidation. Since its creation I have used *Programmer's Dozen* as a teaching aid, a reviewing framework, a conference tutorial and a standalone seminar. It has not escaped my attention – or indeed the attention of publishers, conference goers and colleagues – that the *Programmer's Dozen* would make a good book. Who knows, that may happen.

But I digress. This series of articles exists because, more specifically, PARAMETERIZE FROM ABOVE would make a good article.

Of patterns and recommendations

It may sound like there is a contradiction or conflict in containing PARAMETERIZE FROM ABOVE, which is normally referred to as a pattern, within a set of recommendations. So, is it a pattern or is it a recommendation? It is both: there is no exclusive-or relationship between the two concepts. By definition, a pattern makes a recommendation. A pattern is neither a law nor a hard rule but a recommendation that addresses a particular recurring problem. A pattern is contingent and contextual rather than unconditional and universal. In such cases whether a practice is considered a recommendation or a pattern is mostly a matter of form and presentation. In presenting a pattern, there are matters of style and content that are normally considered important – context, problem forces, solution structure, consequences – that a reader may consider less valuable if they are looking for a more conventional exposition of a practice, one that is perhaps less problem focused and less constrained in its prose and presentation.

The term recommendation also sets a different expectation in the mind of the reader and the listener, one that is perhaps less charged and sometimes more accessible than pattern. Although recommendation needs a little qualification – e.g., weaker than a rule, stronger than a consideration – it is a less polarised and misunderstood term than pattern, which can be appreciated and reasoned about at many different levels and from many different perspectives [Buschmann+2007]. On the other hand, one of the benefits of focusing on a recommendation as a pattern is that it encourages a strong focus on the rationale for the recommendation and the empirical support for it. It is a useful exercise to examine your own development habits and design preferences and the recommendations of others in this light. You may find the process of identifying the nature of the problem, including its context and forces, and weighing up the consequences of the proposed solution, both pros and cons, reveals both surprising insights and contradictions.

Although I have favoured a more conventional presentation of the *Programmer's Dozen* recommendations, the pattern perspective has never been far from my thoughts, especially for PARAMETERIZE FROM ABOVE – and not just because of some nagging sense of guilt that I should write it up! The advice to PARAMETERIZE FROM ABOVE offers an obvious counterbalance to the common alternative – which, for the sake of comparison, we could term HARDWIRE FROM BELOW. The pattern perspective can encourage a more rigorous and balanced assessment of a practice, which is no bad thing when the alternative pattern space is populated and popularised with SINGLETON, MONOSTATE and the assorted application of global and static variables.

Of course, simply because something is made as a recommendation or documented as a pattern, does not mean that it is necessarily good. SINGLETON and co are often expressed as unconditionally good ideas – assuming that pattern → good or 'it's popular, so it must be good' – whereas in practice they are either dysfunctional at heart or dysfunctionally applied [Buschmann+2007]. The benefit of really working through a documented pattern form is that, if studied and applied diligently, limitations and conflicts should become more obvious than is often the case when a recommendation is described in more freeform prose.

When *Programmer's Dozen* was first drawn up I also put together notes for writing up and connecting the recommendations as a pattern language. A pattern language is a collection of patterns that are connected in a form that supports incremental application and growth. However, this work never saw the light of day. In its original form, apart from seeing them as a collection of mutually supporting patterns, the dozen did not obviously form an actual pattern language: the individual patterns were easy to describe, but a simple, non-web-like set of interconnections eluded me. If I were to return to it today, the current ordering of the patterns (as listed earlier) offers a more obvious progression than the order I used five years ago, and is therefore a better pattern sequence around which to form a language. There's a lot more to be said on how to document patterns, pattern sequences and pattern languages, as well as on the role of pattern collections in capturing a particular development style, but that is beyond the scope of the current series of articles [Buschmann+2007]. ■

References

[Buschmann+2007] Frank Buschmann, Kevlin Henney and Douglas C Schmidt, *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*, Wiley, 2007

[Henney2002a] Kevlin Henney, 'Six of the Best', *Application Development Advisor*, May 2002, available from http://www.curbralan.com

[Henney2002b] Kevlin Henney, 'The Rest of the Best', *Application Development Advisor*, June 2002, available from http://www.curbralan.com

[Henney2002c] Kevlin Henney, 'Minimalism: A Practical Guide to Writing Less Code', JAOO, September 2002, available from http://www.curbralan.com

[Henney2007] Kevlin Henney, 'The PfA Papers: From the Top', *Overload 80*, August 2007, http://accu.org/index.php/journals/1411

[Potantin+2002] Alex Potanin, James Noble, Marcus Frean and Robert Biddle, 'Scale-free Geometry in Object-Oriented Programs', *Victoria University of Wellington Computer Science Technical Report*, December 2002, http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-02-30.abs.html, published in Communications of the ACM, May 2005

Blue-White-Red, an Example Agile Process

When it comes to running a project, one size rarely fits all, but Allan Kelly finds that tailoring a core idea over time has led to repeated success.

Blue-White-Red is a simple Agile system originated by Liz Sedley and me for a London company transitioning to Agile development. There wasn't a great deal of up front thought that went into this system, we just started trying to approximate XP [Beck00] and at first it was a poor approximation. We modified our process as we went along. Looking back there is a heavy Scrum [Schwaber02] influence but mainly this is what worked for us.

This process originated at one company and eventually Liz and I left the company and went our separate ways. We both took the same basic process and implemented it elsewhere with modifications. This description also draws on our experience since, so although I've described it as one company it is more of a composite image.

I have come to the conclusion that you can't just take an Agile, or any other, process off the shelf and use it. You have to create a process that works for you. If you do want to take an existing process and implement it then you are going to need help. In fact I would go as far as saying I doubt you can actually implement XP unless you have Kent Beck, Ward Cunningham or one of the other process authors on your team. The same goes for any other process you care to choose, DSDM, Scrum, Crystal, etc.

Basics

The whole system revolved around a large magnetic white board upon which index cards were placed to represent work. The cards themselves were blue, white or red and held on the board with magnets. The board was marked with important information like iteration deadline, a record of how much work was done in previous iterations and was divided into four columns: work to do, in progress, waiting for test and completed work.

Product Managers produced Product Requirements Documents. Pieces of work from these documents, usually features, would be written on blue index cards. The information on these feature cards only needed to be brief, perhaps a title and a document section. These could be prioritised and the highest priority cards looked at in more depth.

According to most Agile methods each card should represent a complete piece of deliverable work. However the nature of our product, the existing code base and perhaps our own inexperience meant that one 'feature' was more work than could be done in a short space of time. So we put the features on blue cards and developers would break the work down into a set of tasks written on white index cards. For every blue feature card there would be multiple white task cards. Each white card could be achieved in a day or two. If it couldn't we tried to break it down further.

Allan Kelly Allan served his apprenticeship developing software for financial, communication and utility systems. He is now a consultant and interim manager who specialises in advising and helping the most challenged development teams deliver and improve. His first book, Changing Software Development, is published by John Wiley and Sons early next year. Allan can be reached at allan@allankelly.net.

The break down from blue to white was usually done during the bi-weekly planning meeting. If the feature was very complicated or poorly understood a special meeting might be held to discuss the work and break it down. Developers could also add white task cards to the work pile if they felt some piece of remedial work was needed, e.g. larger refactorings.

Iterations were two weeks long. They would finish in the morning – a Tuesday, Wednesday or Thursday, never a Monday or Friday. The online systems would complete with a release to live. Then in the afternoon we would convene a planning meeting. (More recently I have been running one week iterations with monthly releases to a live server.)

Planning

In the planning meeting the Product Manager would select the features to be implemented during the next iteration. During the iteration the team would focus on only these features and their associated tasks. Other blue cards would be held offline in an index card box. Since each feature could be quite large there would normally only be a few (one, two or three) feature cards in play at any one time.

Work estimation was done in abstract points. At first this caused some confusion but teams quickly converged on a shared understanding of how much work could be accomplished in a single point. Estimates usually ranged between half a point and two points. Occasionally zero point cards would be written to remind ourselves of things or for trivial tasks.

Although each team placed a slightly different value on a single point we normally found that a card with a point value of more than three needed to be broken down further. We also found that the more words used on a white card to describe the task the more accurate the estimate. Cards with brief descriptions were usually poorly understood and poorly estimated.

The first task in the planning meeting was to clear the board and count the point value of the cards completed in the previous iteration. This was recorded and used as a guide for the coming iteration's capacity. The team could accept slightly more points into the iteration than had been completed the previous week on the understanding that some might not get done.

With blue cards and white cards prepared and our estimate of work that could be done the Product Manager would prioritise all the white cards. Developers would advise of any dependencies between cards, risks, opportunities and such but the final say on prioritisation was the Product Manager's. All cards were prioritised in absolute order – 1, 2, 3, and so on. No two cards were allowed the same priority. This made it clear what was the top priority and which the last. When priorities are set as 'must have', 'should have' and 'nice to have' teams typically end up with too many 'must have's to tackle in one go so the actual work order gets decided by the team. If a business abdicates its responsibility to articulate its needs and priorities to developers then it should not be surprised by the results (although it might be difficult to communicate this message to the business.)

Once work was prioritised the cards could be placed on the board. This formed the work queue. Wherever possible we tried to avoid associating

some developers would cherry pick cards they felt they could do as 'background tasks' or during spare moments

individuals with pieces of work. When someone is named as the individual responsible for a given piece of work the queue does not get worked from top to bottom. Other individuals skip a card which is associated with someone else even if it has a high priority. Of course sometimes you need a particular individual to work on a particular piece of work, but on the whole we tried to avoid this.

Knowing how many points of work the team had completed in previous weeks meant we could be quite confident of what would and would not get done during the coming iteration. Say a team had done 10 points in the previous iteration; we would put about 13 or 14 point on the board for the coming iteration. We could be fairly sure 7 would get done, hopefully another 4 would be cleared and if we were lucky then we might get more.

Developers were not supposed to work on more than one card at a time; this was intended to keep focus. However some developers would cherry pick cards they felt they could do as 'background tasks' or during spare moments. While well intentioned this tended to be disruptive. Developers usually picked refactoring cards and because it was a secondary task it became difficult to track the status. On one occasion I found several cards on a developer's desk, he had intended to do them in 'spare time' but the fact that they were on his desk meant the work was hidden.

Each day the team would hold a short stand up meeting and select the cards they would work on from the work queue. Some developers would choose to pair on some work but we did not pair all the time. If work was completed without pairing it would be subject to a short desk based code review before checking into source code control.

Testing

Developers tried to write unit tests for the new features. However due to the existing legacy code base this was not always possible. There were no unit tests for code that already existed so refactoring existing code was difficult. All unit tests were run each night after the nightly build was completed. Should the build or any tests fail the whole team received mail and the first person in started to investigate the failure.

Each team had a software tester who was responsible for accepting a completed white card. When the developer felt a card was complete it would be moved to the waiting for test column on the whiteboard and the developer would take another card. Only when the tester was satisfied the work was completed would it be marked and moved into the completed column.

Testers had a variety of ways of testing cards: they could perform a manual test, they might ask to verify the unit tests were working and they might ask for proof the code had been reviewed. If they were not satisfied, or a defect was found, then the card would move back to the in progress queue with a high priority.

Finally, if a fault did slip into the system, or was reported by another team and it was added to the team's work load it would be written up on a red card. Red cards automatically took priority as the next piece of work to be started. Unfortunately the nature of the system meant it was difficult to eliminate such tasks but over time the number did fall.

Each team that has used this process has modified it in different ways. No team was able to eliminate all manual testing because it was not possible to retrofit unit tests to parts of the legacy code base. Over time unit test coverage increased but never covered the whole application.

One project that used extensive COM components had particular problems with Test Driven Development (TDD). Testing at the component level tends to be too general, one COM call tends to do too much stuff to test properly, neither is it possible to test the state of the object satisfactory after the COM call. Testing inside the component, below the COM interface tends to be difficult because COM gets into code in all sorts of odd ways. Whether it is memory management, COM pointers, lifetime, startup, or shutdown issues, COM makes it difficult to isolate code for testing.

Micro project variation

I term projects with very little developer resource micro-projects. Typically a micro project has less than two full time developers. Even if there are more than two developers attached to the project, when they are split between multiple projects the net effort may be less than two full time people. For example, developer Fred is full time on the project but Pete is split between three different projects.

One variant of the Blue-White-Red process was used on a one developer micro-project where I played the role of Product Manager with additional responsibility for project management. Here we made several modifications. Firstly the code base was smaller and leant itself more to one feature, one task, one card so we were able to dispense with blue cards altogether and just work with white and the occasional red.

Second, there was no tester on the project and neither had the developer attended the TDD training course our other developers had. So in the last day or two of the iteration development would stop and the developer would test. If necessary, others would join in too. Since this was an online system we could put interim versions of the software onto a staging server during the iteration for preview by customers and feedback.

Finally, there was a pre-planning meeting where the developer and myself (as Product Manager) would get together a few days before the end of the iteration and assess what had been done, what work was coming up, and how long it might take. After this meeting I would be able to decide my priorities prior to the planning meeting

Good luck

What I didn't recognise at the time was how lucky we were to start with. For example, we had source code control and a regular build in place. We couldn't do intra-day builds, it just took too long, but we could know within a day if things were broken.

We were also fortunate that a well-established product owner system was already in place in the form of Product Managers. Although Agile development is good at handling vague and changing requirements you still need someone to articulate what the requirements are and answer questions about how the system would work. In too many organizations developers are left without this guidance and have only their own resources

to fall back on. This can work when a product is mature or when developers are close to the final customers. At other times the business requesting the software seems to rely on direct thought-transference.

Some things we weren't so lucky with. When you start with a large legacy code base, getting unit tests in place is hard. It is not impossible but it is hard. I think there are two main problems here. Firstly there is a mental block: too many developers have a kind of automatic dislike of the word 'test'. 'Testing' is associated with 'software testers' who are obviously paid less and therefore better suited to testing while developers, well, develop.

We had managed to persuade our management that TDD was a good thing and they had paid to bring a trainer in house to deliver a series of training courses. Although most of our developers were trained in TDD some did not feel it was worth doing or believed it took too much time. Unfortunately the management who had paid for the courses declined to make the use of TDD mandatory so usage was patchy in the company apart from the blue-white-red teams.

Still most developers do not have experience of adding tests to legacy code so they simply don't know how to do it. There is one book on the subject [Feathers04] but as good as this is it is no substitute for experience. In retrospect I recognise the need to employ a part-time TDD coach to work with the team in addition to the training.

Retrospectives

The other thing I would do differently is to do more retrospectives. This is hard because we did hold some. Our problem was getting the rest of the company to help implement the recommendations that came out of the retrospective.

Any retrospective will suggest a number of changes in the way people work and the processes followed. When these changes are entirely within the team they are relatively easy to change. But inevitably, over time, these changes are made leaving the more difficult ones to make.

The more difficult changes typically need the involvement of people outside the team and the support of more senior managers. Unless these managers take part in the retrospective it can be hard for them to see the need or opportunity for the change. However trying to persuade a manager to spend several hours in a retrospective is hard.

Like our process our retrospectives were a simplified form and again we used blue, white and red cards. Normally I would start by constructing a timeline [Kerth01]. This is useful because it helps the team remember the early days of the project, without a timeline people tend to concentrate on the most recent events. The time line also helps put the whole project in perspective.

I would put a generous supply of cards on the table and at any time in the retrospective people could write on the card and throw them in a pile in the middle of the table. The rule was:

- Blue cards for thing that worked and we should do again.
- White cards for suggestions for change: these were specific suggestions to do something different.
- Red cards for puzzles, things we don't understand and would like to discuss some more. (Recently I have experimented using Red cards to capture things 'To avoid' doing.)

As we created the timeline (usually on the wall with Post-It notes) people would suggest ideas, write them on cards and throw them into the pile. Once the timeline was built we would walk through it and discuss the events and their sequence. As we did so more cards would be written.

Eventually we would reach a point were we understood the project better. Then it was time to start to wrap up. I would take the cards and sort them into three piles, one for each colour. There was no strict rule but I usually worked through the red cards first. By this stage we had normally answered a lot of the puzzles already. Some of the puzzles would be beyond our understanding and others we could resolve and produce blue and white cards

Next we would start on the blue cards. I would read them out and we would agree (or not) to keep the activity on the card.

Finally the white cards: things to do differently. Quite often people would have suggested the same things. Here it depended on the team and the items in the pile. Some items everyone would agree on and they were within our power to change. Other items we might not agree on, maybe some people would want to change and others would not. Sometimes I would have the team vote on the top three things to change. By limiting the items to change effort can be focused.

Since coming up with this formula Agile Retrospectives [Derby06] has been published. There are more exercises in this book which I will try to incorporate in future retrospectives.

Conclusion

In writing this up I hope to convey a sense of how an Agile process can work, and how you can start with something quite simple and build up. Inevitably I've hidden some details, knocked off some rough edges and highlighted our successes. Simply deciding to follow this, or any other, process doesn't remove all your problems overnight. You still have to work at them.

What this process does do is increase focus and expose problems. Once problems are exposed you can go about fixing them. This is where great improvements can be made. This process provided us with a framework which allowed us to start adopting Agile ideas and to start improving.

Unfortunately exposing problems does not make you popular. Showing the slow pace of development does not look good even if it is true. Exposing problems means someone needs to fix them rather than not ignore them.

The technique I now call Blue-White-Red has worked, with modifications, at three different companies. I don't think this makes it universally applicable but it does show that you can roll-your-own Agile process and I encourage more people to give it a try. ■

References

[Beck00] Beck, K. (2000) *Extreme programming explained*, Addison-Wesley.

[Derby06] Derby, E. and D. Larsen (2006) *Agile Retrospectives*, Pragmatic Programmers.

[Feathers04] Feathers, M. (2004) *Working Effectively with Legacy Code*, Prentice Hall.

[Kelly07] Kelly, A. (2007) *Changing Software Development: Learning to Become Agile*, John Wiley & Sons.

[Kerth01] Kerth, N. L. (2001) *Project Retrospectives*, New York, Dorset House.

[Schwaber02] Schwaber, K. and M. Beedle (2002) *Agile Software Development with SCRUM*.

	Iteration	Retrospective
Blue cards	Feature under development	Things we did right (should do again)
White cards	Development task	Suggestions for improvement
Red cards	Fault (to be fixed as priority)	Puzzles (variation: things to avoid)

Summary of Card Uses

Functional Programming Using C++ Templates (Part 1)

Template metaprogramming can initially seem baffling, but exploring its link to functional programming helps shed some light on things.

Introduction

Computing can be a surprisingly deep field at times. I find that the more I learn about it, the more I'm struck by quite how many similarities there are between different areas of the subject. I was browsing through Andrei Alexandrescu's fascinating book *Modern C++ Design* recently when I read about a connection which I thought was worth sharing.

As I suspect most of you will already be aware, C++ can be used for something called template metaprogramming, which makes use of C++'s template mechanism to compute things at compile time. If you take a look at a template metaprogram, however, you'll find that it looks nothing like a 'normal' program. In fact, anything but the simplest metaprogram can start to look quite intimidating to anyone who's unfamiliar with the idioms involved. This makes metaprogramming seem hard, and can put people off before they've even started.

Surprisingly, the key to template metaprogramming turns out to be functional programming. Normal programs are written in an imperative style: the programmer tells the computer to do things in a certain order, and it goes away and executes them. Functional programming, by contrast, involves expanding definitions of functions until the end result can be easily computed.

Programmers who have studied computer science formally at university are likely to have already come across some form of functional programming, perhaps in a language such as Haskell, but for many self-taught programmers the idioms of functional programming will be quite new. In this article, I hope to give a glimpse of how functional programming works, and the way it links directly to metaprogramming in C++.

For a more detailed look at functional programming, readers may wish to take a look at [Thompson] and [Bird]. Anyone who's interested in template metaprogramming in general may also wish to take a look at the Boost MPL library [Boost]. Finally, for a much deeper look at doing functional programming in C++, readers can take a look at [McNamara].

Compile-time lists

As a concrete example, I want to consider a simple list implementation. For those who are unfamiliar with them, Haskell lists are constructed recursively. A list is defined to be either the empty list, `[]`, or an element (of the appropriate type) prefixed, using the `:` operator, to an existing list. The example `[23,9,84] = 23:[9,84] = 23:9:[84] = 23:9:84:[]` shows how they work more clearly. Working only with lists of integers (Integers in Haskell) for clarity at the moment, we can define the following functions to take the head and tail of a list:

```
head :: [Integer] -> Integer
head (x:xs) = x

tail :: [Integer] -> [Integer]
tail (x:xs) = xs
```

```
template <typename T> struct Head;
template <int x, typename xs>
   struct Head<IntList<x,xs> > {
   enum { value = x };
};

template <typename T> struct Tail;
template <int x, typename xs>
   struct Tail<IntList<x,xs> > {
    typedef xs result;
};
```

Listing 1

The head function takes a list of integers and returns an integer (namely, the first element in the list). The tail function returns the list of integers remaining after the head is removed. So far, so mundane (at least if you're a regular functional programmer).

Now for the interesting bit. It turns out that you can do exactly the same thing in C++, using templates. (This may or may not make you think 'Aha!', depending on your temperament.) The idea (à la Alexandrescu) is to store the list as a type. We declare lists of integers as follows:

```
struct NullType;
template <int x, typename xs> struct IntList;
```

The `NullType` struct represents the empty list, `[]`; the `IntList` template represents non-empty lists. Using this scheme, our list `[23,9,84]` from above would be represented as the type `IntList<23, IntList<9, IntList<84, NullType> > >`. A key point here is that neither of these structs will ever be instantiated (that's why they're just declared rather than needing to be defined): lists are represented as types here rather than objects.

Given the above declarations, then, we can implement our head and tail functions as shown in Listing 1.

Already some important ideas are emerging here. For a start, if we ignore the fact that the C++ version of the code is far more verbose than its Haskell counterpart (largely because we're using C++ templates for a purpose for which they were never designed), the two programs are remarkably similar. We're using partial template specialization in C++ to do the job done by pattern-matching in Haskell. Integers are being defined using

Stuart Golodotz Stuart has been programming for 13 years and is currently studying for a computing doctorate at Oxford University. His next project involves the geometric modelling of kidney cancer. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

the analogy between functional programming in Haskell and compile-time programming in C++ is extremely deep

enums and lists are defined using **typedef**s (remember once again that lists are represented as types).

Using these constructs is rather clumsy. A program outputting the head of the list **[7,8]**, for example, currently looks like:

```
#include <iostream>

int main() {
  std::cout << Head<IntList<7,IntList<8,
               NullType> > >::value << std::endl;
  return 0;
}
```

To improve this sorry state of affairs, we'll use macros (this is one of those times when the benefits of using them outweigh the disadvantages). In a manner analogous to that used for 'typelists' in *Modern C++ Design*, we define the macros in Listing 2 to help with list creation.

We also define macros for head and tail:

```
#define HEAD(xs)                Head<xs>::value
#define TAIL(xs)                Tail<xs>::result
```

The improvement in the readability and brevity of the code above is striking:

```
std::cout << HEAD(INTLIST2(7,8)) << std::endl;
```

From now on, we will assume that when we define a new construct, we will also define an accompanying macro to make it easier to use.

Outputting a list

Before implementing some more interesting list algorithms, it's worth briefly mentioning how to output a list. It should come as no surprise that the form of our output template differs from the other code in this article: output is clearly done at runtime, whereas all our other list manipulations are done at compile-time. We can output lists using the code in Listing 3.

```
template <typename T> struct OutputList;

template <> struct OutputList<NullType> {
  void operator()() {
     std::cout << "Null" << std::endl;
  }
};

template <int x, typename xs>
   struct OutputList<IntList<x,xs> > {
  void operator()() {
    std::cout << x << ' ';
    OutputList<xs>()();
  }
};
```

Listing 3

Sorting

Computing the head and tail of a list constructed in a head:tail form may seem a relatively trivial example. Our next step is to try implementing something a bit more interesting: sorting. Perhaps surprisingly, this isn't actually that difficult. The analogy between functional programming in Haskell and compile-time programming in C++ is extremely deep, to the extent that you can transform Haskell code to C++ template code almost mechanically. For this article, we'll consider two implementations of sorting, selection sort and insertion sort (it would be just as possible, and not a great deal harder, to implement something more efficient, like quicksort: I'll leave that as an exercise for the reader). I've confined my implementation to ordering elements using **operator<**, but it can be made more generic with very little additional effort.

Selection sort

A simple selection sort works by finding the minimum element in a list, moving it to the head of the list and recursing on the remainder. We're thus going to need the following: a way of finding the minimum element in a list, a way of removing the first matching element from a list and a sorting implementation to combine the two. Listing 4 shows how we'd do it in Haskell.

We can transform this to C++ as shown in Listing 5.

The important things to note here are that each function in the Haskell code corresponds to a C++ template declaration, and each pattern-matched case in the Haskell code corresponds to a specialization of one of the C++ templates.

```
#define NULLLIST                NullType
#define INTLIST1(n1)            IntList<n1, NULLLIST>
#define INTLIST2(n1,n2)         IntList<n1, INTLIST1(n2)>
#define INTLIST3(n1,n2,n3)      IntList<n1, INTLIST2(n2,n3)>
#define INTLIST4(n1,n2,n3,n4)   IntList<n1, INTLIST3(n2,n3,n4)>
...
```

Listing 2

we need to generate one of two different types depending on the value of a boolean condition, which is non-obvious

```
minElement :: [Int] -> Int
minElement [m] = m
minElement (m:ms) = if m < least then m else least
   here least = minElement ms

remove :: Int -> [Int] -> [Int]
remove n (n:ms) = ms
remove n (m:ms) = m : (remove n ms)

ssort :: [Int] -> [Int]
ssort [] = []
ssort ms = minimum : ssort remainder
   where minimum = minElement ms
         remainder = remove minimum ms
```

Listing 4

Insertion sort

Implementing insertion sort is quite interesting. The essence of the algorithm is to insert the elements one at a time into an ordered list, preserving the sorted nature of the list as an invariant.

A simple Haskell implementation of this goes as follows:

```
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (x:xs) = if n < x
   then n:x:xs else x:(insert n xs)

isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Translating the `insert` function to C++ is not entirely trivial. The problem is that we need to generate one of two different types depending on the value of a boolean condition, which is non-obvious. There are (at least) two solutions to this: we can either rewrite the Haskell function to avoid the situation, or we can write a special C++ template to select one of two `typedef`s based on a boolean condition.

Rewriting the Haskell code could be done as follows:

```
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (x:xs) = smaller : (insert larger xs)
   where (smaller,larger) = if n < x then (n,x)
   else (x,n)
```

This solves the problem (generating one of two different values depending on the value of a boolean condition is easy), but at the cost of a less efficient function.

```
// Finding the smallest element of a list
template <typename T> struct MinElement;
template <int x>
   struct MinElement<IntList<x,NullType> > {
  enum { value = x };
};
template <int x, typename xs>
   struct MinElement<IntList<x,xs> > {
  enum { least = MinElement<xs>::value };
  enum { value = x < least ? x : least };
};

// Removing the first element with a given value
// from a list
template <int n, typename T> struct Remove;
template <int n, typename xs> struct Remove<n,
   IntList<n,xs> > {
  typedef xs result;
};
template <int n, int x, typename xs>
   struct Remove<n, IntList<x,xs> > {
  typedef IntList<x,
     typename Remove<n,xs>::result> result;
};

// Sorting the list using selection sort
template <typename T> struct SSort;
template <> struct SSort<NullType> {
  typedef NullType result; };
template <int x, typename xs>
   struct SSort<IntList<x,xs> > {
  enum {
    minimum = MinElement<IntList<x,xs> >::value };
  typedef typename Remove<minimum,
     IntList<x,xs> >::result remainder;
  typedef IntList<minimum,
     typename SSort<remainder>::result> result;
};
```

Listing 5

The template version (using the `Select` template borrowed directly from Andrei's book) does a better job:

```
template <bool b, typename T, typename U>
   struct Select {
  typedef T result;
};
template <typename T, typename U>
   struct Select<false, T, U> {
  typedef U result;
};
```

we can make lists of anything that can be represented by integers at compile-time

```
// Inserting a value into an ordered list
template <int n, typename T> struct Insert;

template <int n> struct Insert<n, NullType> {
  typedef IntList<n, NullType> result;
};

template <int n, int x, typename xs>
   struct Insert<n, IntList<x,xs> > {
  typedef IntList<n, IntList<x,xs> > before;
  typedef IntList<x,
     typename Insert<n,xs>::result> after;
  typedef typename Select<(n < x), before,
     after>::result result;
};

// Sorting the list using insertion sort
template <typename T> struct ISort;

template <> struct ISort<NullType> {
  typedef NullType result; };

template <int x, typename xs>
   struct ISort<IntList<x, xs> > {
  typedef typename Insert<x,
     typename ISort<xs>::result>::result result;
};
```

Listing 6

This allows us to straightforwardly transform the more efficient form of the Haskell code to C++ (Listing 6).

```
template <int n, int x, typename xs>
   struct InsertBefore {
  typedef IntList<n, IntList<x,xs> > result;
};

template <int n, int x, typename xs>
   struct InsertAfter {
  typedef IntList<x,
     typename Insert<n,xs>::result> result;
};

template <int n, int x, typename xs>
   struct Insert<n, IntList<x,xs> > {
  typedef InsertBefore<n,x,xs> before;
  typedef InsertAfter<n,x,xs> after;
  typedef typename Select<(n < x), before,
     after>::result::result result;
};
```

Listing 7

It turns out that in C++ this still isn't as efficient as it could be. The culprit is in the second specialization of **`Insert`** – by defining the **`before`** and **`after`** **`typedef`**s in the specialization itself, we force them both to be instantiated even though only one is actually needed. The solution is to introduce an extra level of indirection (Listing 7).

This solves the problem, because now the chosen **`IntList`** template only gets instantiated if it is actually needed.

Maps and filters

One of the best things about writing in a functional language has traditionally been the ability to express complicated manipulations in a simple fashion. For example, to apply the same function **`f`** to every element of a list **`xs`** in Haskell is as simple as writing **`map f xs`**. Similarly, filtering the list for only those elements satisfying a boolean predicate **`p`** would simply be **`filter p xs`**. A definition of these functions in Haskell is straightforward enough:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : remainder
  else remainder
  where remainder = filter p xs
```

Achieving the same thing in C++ initially seems simple, but is actually slightly subtle. The trouble is in how to define **`f`** and **`p`**. It turns out that what we need here are template template parameters. Both **`f`** and **`p`** are template types which yield a different result for each value of their template argument. For instance, a 'function' to multiply by two could be defined as:

```
template <int n> struct TimesTwo {
  enum { value = n*2 };
};
```

and a predicate which only accepts even numbers could be defined as

```
template <int n> struct EvenPred {
  enum { value = (n % 2 == 0) ? 1 : 0 };
};
```

The **`Map`** and **`Filter`** templates can then be defined as in Listing 8.

Note that we again make use of the **`Select`** template to choose between the two different result types in **`Filter`**.

we need to generate one of two different types depending on the value of a boolean condition, which is non-obvious

```
template <template <int> class f,
   typename T> struct Map;
template <template <int> class f>
   struct Map<f, NullType> {
  typedef NullType result;
};

template <template <int> class f, int x,
   typename xs>
struct Map<f, IntList<x, xs> > {
  enum { first = f<x>::value };
  typedef IntList<first,
     typename Map<f,xs>::result> result;
};

template <template <int> class p, typename T>
   struct Filter;
template <template <int> class p>
   struct Filter<p, NullType> {
  typedef NullType result;
};

template <template <int> class p, int x,
   typename xs>
struct Filter<p, IntList<x,xs> > {
  enum { b = p<x>::value };
  typedef typename Filter<p,xs>::result remainder;
  typedef typename Select<b, IntList<x,remainder>,
     remainder>::result result;
};
```

Listing 8

Extensions

So far, we've only seen how to implement integer lists. There's a good reason for this – things like doubles, for example, can't be template parameters. All isn't entirely lost, however. It turns out that we can make lists of anything that can be represented by integers at compile-time! The code looks something like Listing 9.

The important change is in how we treat the head of the list – now we write `typename x` wherever we had `int x` before, and use the type's value field to get its actual value if we need it. The rest of the code can be transformed to work for generic lists in a very similar fashion. There's something to be said about how we handle ordering, but that's a topic for the next article!

Conclusion

In this article, we've seen how template metaprogramming is intrinsically related to functional programming in languages like Haskell, and implemented compile-time lists using C++ templates. Next time, I'll show one way of implementing ordering in generic lists, and consider how to implement compile-time binary search trees.

```
template <int n> struct Int {
  typedef const int valueType;
  static valueType value = n;
};

template <int n, int d> struct Rational {
  typedef const double valueType;
  static valueType value;
};

template <int n, int d> const double
   Rational<n,d>::value = ((double)n)/d;

template <typename T> struct Head;
template <typename x, typename xs>
   struct Head<List<x,xs> > {
  typedef x result;
};

#define HEAD(xs)Head<xs>::result::value
```

Listing 9

So what are the uses of writing code like this? One direct use of compile-time BSTs would be to implement a static table that is sorted at compile time. This can prove extremely helpful, particularly in embedded code. There are also indirect benefits derived from learning more about template metaprogramming in general. Writing code like this can be seen as a useful stepping stone towards understanding things like the typelists described in Andrei's book. The capabilities these provide are quite astounding and can provide us with real benefits to the brevity and structure of our code.

Till next time... ■

Acknowledgements

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

References

[Bird] *Introduction to Functional Programming*, Richard Bird and Philip Wadler, Prentice Hall

[Boost] http://www.boost.org/libs/mpl/doc/index.html

[McNamara] *Functional Programming in C++*, Brian McNamara and Yannis Smaragdakis, ICFP '00

[Thompson] Haskell: *The Craft of Functional Programming*, Simon Thompson, Addison Wesley

auto_value: Transfer Semantics for Value Types

In his quest to pass around values efficiently, Richard Harris concludes by looking at ways of transferring ownership, now and in the future.

Last time we took a look at string and discussed the implications of the copy-on-write, or COW, optimisation. I concluded that despite its many shortfalls, I was unwilling to dismiss this type of optimisation out of hand.

This time, I'll explain why.

auto_string

Recall that it's **auto_ptr** rather than **shared_ptr** that is designed to manage ownership transfer. So, why not consider **auto_ptr** rather than **shared_ptr** semantics to eliminate the copy?

Let's look at a slightly different definition of **string** (Listing 1).

```
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(const auto_string &s);

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

  auto_string release();

  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.

```
class auto_string
{
  friend class string;

  auto_string(string::size_type n,
              const auto_array<char> &s) throw();

  string::size_type size_;
  auto_array<char> data_;
};
```

Listing 2

```
auto_string
  string::release()
  {
    return auto_string(size_, data_.release());
  }
```

Listing 3

Here, the data is stored in a scoped_array (like **scoped_ptr** but uses **delete[]**) rather than a **shared_array** and we've picked up a few extra functions, all of which refer to a new type, **auto_string**.

Let's have a look at the definition of **auto_string** (Listing 2).

So, **auto_string** is simply a wrapper for an **auto_array** (which also uses **delete[]**) that hides its data from everything but **string**.

This gives us an explicit mechanism for stating that we wish to transfer ownership of a **string**, through the **release** member function (Listing 3).

The constructor and assignment operator can now be overloaded to take advantage of the fact that we are transferring ownership (Listing 4).

Now, when we assign an **auto_string** returned from a function to a **string** (Listing 5), we have the sequence of events shown in Listing 6 and at no point is the string data copied.

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

string &
string::operator=(const auto_string &s)
{
  string tmp(s);
  swap(tmp);
  return *this;
}
```

Listing 4

at every step of copy construction or assignment, the string data is owned by one and only one object

```
auto_string
f()
{
  string result("hello, world");
  return result.release();
}

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

Listing 5

```
call g
default construct s;

call f
construct "hello, world" into result
transfer s into temporary auto_string
transfer temporary auto_string into return value
exit f

transfer auto_string into tmp
swap data with tmp
exit g
```

Listing 6

The advantage this has over the **shared_array** version is that at every step of copy construction or assignment, the string data is owned by one and only one object. In other words, we no longer have aliasing of the data and consequently don't have to deal with the headaches that that causes.

Of course, since the compiler can optimise the copy away anyway, all we have succeeded in doing is to create a slightly better version of a completely pointless optimisation.

Well, not quite.

Thank you for your patience, by the way.

vector

There is, in fact, another benefit to explicit lifetime control but this is much better illustrated with a class representing a mathematical vector.

To start, let's have a brief recap of **valarray**. Just kidding – see Listing 7.

The principal difference between this and a **std::vector<double>** is that we want to support mathematical vector operations.

Let's use vector addition as an example (Listing 8).

Now this isn't the most efficient implementation, creating an uninitialised vector and filling it with the result of the addition would have 3n read and write operations whereas this has 4n. Still, it does enable us to reuse the in-place addition operator.

Note that we are assuming that both in-place addition and out-of-place addition require 2n operations, with out-of-place addition incurring a further n to copy the results. Strictly speaking this is not the case since the processor must make 3n reads and writes from main memory in both cases.

However, for many processors in-place addition will make much better use of the processor cache and as a result will generally be more efficient.

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

  vector();
  vector(const vector &v);
  vector(const auto_vector<T> &v);
  explicit vector(size_t n);

  vector & operator=(const vector &v);
  vector & operator=(const auto_vector &v);

  auto_vector release();

  const_iterator begin() const;
  const_iterator end() const;
  iterator begin();
  iterator end();
  //...
  vector& operator+=( const vector& v );
  //...

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

Listing 7

```
vector
operator+(const vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return tmp;
}
```

Listing 8

it allows us to indicate when we no longer care what happens to an object

```
vector
operator+(vector l, const vector &r)
{
  l += r;
  return l;
}
```

Listing 9

```
auto_vector operator+(const vector &l,
                      const vector &r);
auto_vector operator+(const auto_vector &l,
                      const vector &r);
auto_vector operator+(const vector &l,
                      const auto_vector &r);
auto_vector operator+(const auto_vector &l,
                      const auto_vector &r);
```

Listing 10

```
auto_vector
operator+(const auto_vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return tmp.release();
}
```

Listing 11

A few crude experiments with my compiler supported this, showing that out-of-place addition did indeed take approximately 1.5 times as long as in-place addition, so we shall maintain these complexity assumptions as a convenient fiction.

We can effectively address the efficiency concern when `l` is bound to a function return value by redefining the operator as shown in Listing 9.

Now we are exploiting copy elision to reduce the number of reads and writes to 2n, even better than if we were to use an uninitialised vector to store the result.

Unfortunately, we'll still pay for the extra copy when `l` is a variable. This is especially irritating if `r` is a function return value and hence a candidate for in-place addition. Since reference to `T` and value of `T` are afforded the same status during function overload resolution we can't use overloading to address this. Unless we use a different type for our temporaries. Like `auto_vector`, for example (Listing 10).

The implementation of each overload will look like our first version, except that we will prefer to construct the temporary from an `auto_vector` whenever possible, so that we can avoid copying one of the arguments (Listing 11).

So the long awaited advantage of this approach is that it allows us to indicate when we no longer care what happens to an object, which we can exploit both for transfer initialisation and for transforming out-of-place operations into in-place operations.

```
auto_vector
f()
{
  vector x;
  //...
  return x.release();
}

auto_vector
g()
{
  vector x;
  //...
  return x.release();
}

auto_vector
h()
{
  vector x;
  //...
  return x.release();
}

void
i()
{
  vector sum = f()+g()+h();
}
```

Listing 12

Note that by returning `auto_vectors` from the addition operators we can eliminate the construction of a series of temporaries in a compound expression (Listing 12).

With the original implementation of addition, the cost of the summation would have been:

- 2 copies to temporary values at 2n read/writes each
- 2 in-place additions at 2n read/writes each

This gives us a grand total of 8n read/write operations.

Recall that if we were willing to forego reuse of the in-place addition operator, we could remove one of the read/write operations from creating each of the temporary values, making the cost of summation:

- 2 additions at 2n read/writes each
- 2 copies to temporary values at n read/writes each

Reducing the total to 6n read/write operations.

With `auto_vector`, however, this becomes:

- 5 transfers to temporary values at O(1) read/writes each

```
double
abs(const vector &v)
{
  double sum_square = 0.0;
  vector::const_iterator first = v.begin();
  vector::const_iterator last  = v.end();

  while(first!=last)
  {
    sum_square += *first * *first;
    ++first;
  }

  return sqrt(sum_square);
}
```

Listing 13

- 2 in-place additions at 2n read/writes each

Giving us a final total of 4n + O(1) read/write operations. Not too shabby.

There are two principal disadvantages to this approach.

Firstly we have to write overloaded versions of every function and secondly we have to write a lot of boilerplate code.

Don't we?

The crux of the first problem is that we only want to overload a function if there is an advantage to us to do so. In other words, we only want to overload functions which can exploit the reuse of temporary objects. A lot of functions can't and we really don't want to make work for ourselves by having to overload them. See Listing 13, for example.

Thankfully, in such cases it's perfectly OK to pass an **auto_vector** instead of a **vector**. This is because the compiler is allowed to bind a const reference to a temporary that is the result of a conversion. And a **vector** can be conversion constructed from an **auto_vector**.

The second problem is a little trickier to resolve.

auto_value

What we really need is a class that can manage the value transfer semantics for us. Much like **auto_ptr** does for pointers.

The class definition is actually pretty straightforward (Listing 14).

It's the implementation that's the problem.

What we really need to find is a generic way to transfer ownership of the controlled value to and from the **auto_value**.

Fortunately, for most of the types we are interested in, one already exists in **swap**.

Let's see how we can use it to implement the member functions of **auto_value** (Listing 15).

```
template<typename X>
class auto_value
{
public:
  typedef X element_type;

  explicit auto_value(X &x) throw();
  auto_value(const auto_value &x) throw();

  auto_value & operator=(const auto_value &x)
throw();

  X & get() const throw();

private:
  mutable X x_;
};
```

Listing 14

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  x_.swap(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  x_.swap(x.get());
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  x_.swap(x.get());
  return *this;
}
```

Listing 15

The chief problem with using **swap** rather than transferring the state directly is that we have to default construct a value before we can swap the transferred value in. This pretty much limits **auto_value** to types with relatively inexpensive default constructors.

Note that we've supplied an explicit transfer constructor for the original value. This is more in keeping with the **auto_ptr** interface and removes the need to add a **release** method to the class.

There's not much we can do about the conversion constructor and assignment operator that need to be implemented in the class itself.

Fortunately, these are pretty simple (Listing 16).

We could make it easier still if we were to abandon our sensibilities and define these functions inline using a macro.

I can't quite bring myself to write it though.

There's only one thing left that's really lacking from the **auto_value** interface and that's **operator.**. This would allow us to use the **auto_value** as a proxy for calls to the transferred object's member functions in the same way that **operator*** and **operator->** do for **auto_ptr**.

Shame it doesn't exist.

Fortunately, there's another way to do this.

That other way is inheritance. If our **auto_value** were to inherit from rather than contain the value it controls, it would trivially be able to act as a proxy.

Let's have a look at the changes we need to make (Listing 17).

The unfortunate side effect of this is that the **auto_value** and the value it controls are now the same entity, and hence a const **auto_value** implies a const value. Now, you'll recall that function return values can't be bound to non-const references and that **swap** is a non-const member function that has a non-const reference argument.

```
X::X(const auto_value<X> &x)
{
  swap(x.get());
}

X &
X::operator=(const auto_value<X> &x)
{
  swap(x.get());
  return *this;
}
```

Listing 16

```
template<typename X>
class auto_value : public X
{
public:
  typedef X element_type;

  explicit auto_value(X &x) throw();
  auto_value(const auto_value &x) throw();

  auto_value & operator=(
     const auto_value &x) throw();
};
```

Listing 17

That's right. We can't use **swap** to implement our transfer semantics any more. Not unless we cast away the constness, that is (Listing 18).

Pretty slick, I'm sure you'll agree.

There's just one tiny little problem with this code. Hardly even worth mentioning.

It's implementation-defined whether or not this will invoke the dreaded undefined behaviour.

There's a clause in the C++ standard stating that compilers are allowed to bind const references to function returns in one of two ways. They can either bind to the value itself, or they can copy it into a const value and bind to that.

Seems innocuous enough, but there's another clause that states that trying to modify a const value results in undefined behaviour. Which we all know could mean anything from doing exactly what you expect to washing away our coding sins with the cleansing fire of a thermonuclear explosion.

I can't think of any compiler vendors who'd go for the latter option though.

Well, on reflection…

Actually, no, not even them.

transfer

Fortunately we can avoid invoking undefined behaviour if we are willing to force users of **auto_value** to do a little additional work.

Specifically, we'll need them to implement an ownership transfer mechanism that will work for const objects. Let's call it **transfer** and have a look at how vector might implement it (Listing 19).

Firstly, we've added a typedef that defines **auto_vector** as an **auto_value** of vector. Secondly, we've added a protected member function transfer to manage the ownership transfer. Finally, we've made the **data_ member** mutable so that the const transfer function can affect it.

A protected member function? Yeah, yeah, I know.

If you're really bothered by it you can grant friendship to **auto_value<vector>** and make it private instead. The point is that the **transfer** member must be accessible by the derived **auto_value<vector>**, but probably shouldn't be public since it will violate the perfectly reasonable expectation that const objects won't change.

Some of you may also balk at the thought of forcing client classes to make their state mutable.

Well, I can give two reasons why you shouldn't worry too much about it.

Firstly, we have no choice. By making the state mutable we are allowed to change it even if the object is *really* const. This enables us to neatly side step the clause in the standard that allows compilers to copy a return value into a const object.

Secondly, it doesn't matter. Const methods are already allowed to change the state of the object.

You may find the second point a little surprising, so I'll elaborate.

The **vector** class, like most container types, stores its state in a memory buffer. If you took a look at the definition of **scoped_array**, you'd notice

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  swap(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  swap(const_cast<auto_value &>(x));
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  swap(const_cast<auto_value &>(x));
  return *this;
}
```

Listing 18

that the access member functions are all const, but return non-const pointers or references. This means that the elements of the array can be changed even when accessed through a const method. Whilst this may seem a little counter intuitive, it's exactly how a pointer data member would behave. They key point to note is that constness doesn't propagate through the pointer.

Namely:

```
X * const x;
```

and:

```
X const * const x;
```

do *not* mean the same thing. The former defines an immutable pointer to a mutable value and the latter an immutable pointer to an immutable value, and it is the former that is implicitly applied to pointer data members in a const member function.

The behaviour we've come to expect from our container classes, that const member functions will not change the value of any items in the container, is enforced by the programmer rather than the compiler. When implementing container classes we must always be careful not to change the state through const member functions since the compiler is unlikely to warn us that we are doing so. Making the state mutable therefore has little effect on the effort we must spend designing the class.

Now we're finished discussing the design choices, let's take a look at the implementation:

```
void
vector::transfer(const auto_value<vector> &v) const
{
  data_.swap(v.data_);
}
```

So **transfer** is actually just a **swap** for const objects, which shouldn't be particularly surprising since we've been using **swap** for ownership transfer from the start.

We'll need to provide new implementations of the conversion constructor and assignment operator used for ownership transfer (Listing 18).

Again, these shouldn't be particularly surprising. We've just replaced the calls to **swap** with calls to **transfer**.

Finally, we need to rewrite the **auto_value** member functions to make use of the **transfer** function (Listing 19).

Yet again, we have simply replaced the calls to **swap** with calls to **transfer**.

So, there we have it, a simple class implementing explicit ownership transfer that requires just a few trivial member functions and a relaxed attitude to constness in its client classes.

```
class vector
{
public:
  typedef double              value_type;
  typedef double *            iterator;
  typedef double const *      const_iterator;
  typedef size_t              size_type;
  typedef auto_value<vector> auto_vector;
  //...

  vector();
  vector(const vector &v);
  vector(const auto_vector<T> &v);
  explicit vector(size_t n);

  vector & operator=(const vector &v);
  vector & operator=(const auto_vector &v);

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

protected:
  void transfer(const auto_vector &v) const;

private:
  mutable scoped_array<double> data_;
};
```

Listing 19

If we are willing to accept this compromise, **auto_value** provides the same functionality as a custom made value transfer type with a lot less work.

Recalling the **auto_vector** addition operators (Listing 20).

We can implement these operators in the same way as before, by constructing a temporary vector from an **auto_vector** argument (where there is one) and performing the addition in-place (Listing 21).

In fact, since the **auto_value** copy constructor transfers ownership, we can further simplify these operators by passing the **auto_value** arguments by value. For example:

```
auto_vector operator+(auto_vector l,
                      const vector &r);
```

Now, instead of transferring the **auto_vector** into a temporary vector, we can exploit the fact that **auto_vector** inherits from **vector** to perform in-place addition directly (Listing 22), saving us a little bit of typing and a few calls to transfer.

Once again, if we don't care about reusing temporaries we simply provide a single version of a function:

```
double abs(const vector &v);
```

As before, this function will work for both vectors and **auto_vectors** although now this is because **auto_vector** inherits from **vector** and can therefore be bound directly to the reference.

```
vector::vector(const auto_value<vector> &v)
{
  transfer(v);
}

vector &
vector::operator=(const auto_value<vector> &v)
{
  transfer(v);
  return *this;
}
```

Listing 20

```
template<typename X>
auto_value<X>::auto_value(X &x)
{
  transfer(x);
}

template<typename X>
auto_value<X>::auto_value(const auto_value &x)
{
  transfer(x);
}

template<typename X>
auto_value<X> &
auto_value<X>::operator=(const auto_value &x)
{
  transfer(x);
}
```

Listing 21

namespace mojo

In his article 'Generic<Programming>: Move Constructors' (*Dr Dobb's Journal*, 2003), Alexandrescu describes the Mojo framework for automatically detecting temporaries and using move semantics for them.

By its very nature this technique must rely upon implicit, rather than explicit, conversion to transfer types and hence requires a little extra work to ensure that temporaries are bound to the correct transfer type.

The upshot of this is that three overloads, rather than two, must be provided for each function that exploits move semantics. Nevertheless, this is arguably a more sophisticated solution to the problem.

T &&t

To close, it's worth mentioning that the value transfer semantics we've worked so hard to implement are trivial using the above notation.

If you're wondering why on Earth I didn't just go ahead and use it instead of leading you down the garden path, it's because it isn't C++. Not yet.

The notation is for a new kind of reference proposed for the next version of C++, the rvalue reference.

The rvalue reference differs from the familiar reference (hereafter known as an lvalue reference) in eight ways. Paraphrasing the proposal slightly:

1. A non-const rvalue can bind to a non-const rvalue reference.
2. Overload resolution rules prefer binding rvalues to rvalue references and lvalues to lvalue references.
3. Named rvalue references are treated as lvalues.
4. Unnamed rvalue references are treated as rvalues.
5. The result of casting an lvalue to an rvalue reference is treated as an rvalue.
6. Where elision of copy constructor is currently allowed for function return values, the local object is implicitly cast to an rvalue reference.
7. Reference collapsing rules are extended to include rvalue references.
8. Template deduction rules allow detection of rvalue/lvalue status of bound argument.

```
typedef auto_value<vector> auto_vector;

auto_vector operator+(const vector &l,
                      const vector &r);
auto_vector operator+(const auto_vector &l,
                      const vector &r);
auto_vector operator+(const vector &l,
                      const auto_vector &r);
auto_vector operator+(const auto_vector &l,
                      const auto_vector &r);
```

Listing 22

```
auto_vector
operator+(const auto_vector &l, const vector &r)
{
  vector tmp(l);
  tmp += r;
  return auto_vector(tmp);
}
```

Listing 23

For the purpose of eliminating copies, behaviours 1-3 are of particular interest. Put simply, they mean that a non-const temporary *can* be bound to a reference type through which we can legally modify them. Specifically, we now have four kinds of reference (Listing 23).

In the same way that a non-const object is preferentially bound to a non-const reference, an rvalue is preferentially bound to an rvalue reference and an lvalue is preferentially bound to an lvalue reference.

Given the following function signatures:

```
f(T &&t);        //1
f(T const &&t); //2
f(T &t);         //3
f(T const &t);  //4
```

and the following references:

```
T &&t1;
T const && t2;
T & t3;
T const & t4;
```

The rules mean that:

t1 binds to overloads 1, 2, 3 and 4 in that order of preference
t2 binds to overloads 2 and 4 in that order of preference
t3 binds to overloads 3, 4, 1 and 2 in that order of preference
t4 binds to overloads 4 and 2 in that order of preference

The original justification for disallowing the binding of rvalues to non-const references was that it was generally a mistake to do so. Changing an object which is about to go out of scope and be destroyed will, after all, lose those changes.

This is especially nasty when the object in question is a temporary resulting from an implicit conversion (Listing 24).

Since the character `c` is promoted to a long with an implicit conversion, `f` receives a reference to the temporary long that is the result of that conversion. The upshot of which is that, generally to the surprise and consternation of the programmer, `c` never changes.

Move semantics, however, have brought to light a situation in which this is a valid thing to want to do. The new rvalue references provide a

```
auto_vector
operator+(auto_vector l, const vector &r)
{
  l += r;
  return l;
}
```

Listing 24

```
//non-const reference to rvalue
typedef T &&       Ref1;

//const reference to rvalue
typedef T const && Ref2;

//non-const reference to lvalue
typedef T &        Ref3;

//const reference to lvalue
typedef T const &  Ref4;
```

Listing 25

mechanism by which we can express that we deliberately wish to change the value of a temporary object whilst avoiding the original problem.

Specifically, overloading on non-const rvalue reference and const lvalue reference enables us to distinguish between destructively reusing a temporary object and non-destructively referring to a non-temporary object.

Which is, of course, exactly what we've been striving to achieve.

article::~ article()

So has this all been a tremendous waste of time?

I hope not.

Firstly, I hope that this has given you cause to think a little more about how you can exploit ownership control.

Secondly, I hope that you may find `auto_value`, or something like it, a useful stop gap.

And finally, I hope that you've enjoyed it.

If not, I refer you to Harris's Addendum:

Nyah, nyah. I can't hear you. ■

#include

[Alexandrescu, 2003] Alexandrescu, 'Generic<Programming>: Move Constructors' (*Dr Dobb's Journal*, 2003).

[Hinnant, 2004] Hinnant, Abrahams and Dimov, 'A Proposal to Add an Rvalue Reference to the C++ Language' (ISO/IEC JTC1/SC22/WG21 Document Number N1690, 2004).

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

```
void f(long &i);

void g()
{
  char c = 0;
  f(c); //oops
  //...
}
```

Listing 26