contents

credits & contacts

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

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

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Thaddaeus Frogley
ads@accu.org

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

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: Need to Unlearn

Like many men I tend to like my old clothes. Unlike many I've come up with a strategy that helps me enjoy buying new ones: I clear out the wardrobe; rather ruthlessly I throw away things and make space. This done I know I need new clothes and can enjoy buying them.

I need to use a similar strategy when I write articles and patterns. Sometimes I draft a piece and there is a sentence or even a paragraph which I really like, maybe it's witty, sarcastic or makes a subtle side point, or maybe it's an excellent example of something. So, I edit my draft – and I should mention I do a lot of editing – and as I edit I keep the chosen sentence. But over time it doesn't connect as well with what is around it and maybe I have to rewrite some of the surrounding text to lead up to this one sentence.

Eventually it becomes clear that this sentence is more of an obstruction than a support. It has to be chopped out so the rest of the text can make its point with brevity and clarity. It may be painful to do – like getting rid of those old clothes – but you're better off without it.

(Actually, in truth, I usually use the same trick I do with old trousers. Take them out of the wardrobe and put them to one side somewhere, usually in a box under the bed. If you don't need it in the next six months you aren't ever going to need it. So, I find files on my hard disc with little bits of articles which I never get around to using.)

The same thing is true in software. Sometimes a piece of code is so attractive I don't want to lose it – say it's a nifty bit of template meta-programming, or a well-formed class. No matter how nifty the code it can still restrain you, sometimes these things have to go for the greater good.

And it doesn't end with code. In fact, those who study these things would consider this *unlearning.* In the same way that we learn something we sometimes need to *unlearn* something. A solution that worked well in the past doesn't work well now. If we continue to rely on yesterday's solutions we stop ourselves from learning new things, like clothes our solutions come to look dated and full of holes.

The software development community could benefit from a bit more unlearning. While we're pretty good at dreaming up new languages and methods we're not so good at throwing some old ideas away. Sometimes our old ideas work to our benefit, they allow us to quickly diagnose problems and develop solutions because we've seen the problem before.

On other occasions these very shortcuts work against us. We use mental models and assumptions that aren't valid for the problem in hand. Worst of all, we don't always know we're making these assumptions. When I was an undergraduate I had a lecturer who always told us to "Document your assumptions." Problem was, I didn't realise that I was making assumptions. That's one of the problems we face, unconscious assumptions, how do we know we are making them?

Sometimes of course there are big red flags telling us to drop our assumptions. For example, when you change jobs, in a different company with different people we need to change. Unfortunately it's too easy to keep fighting the last war, or see our last employer through rose-tinted spectacles, your new colleagues don't necessarily want to hear about how good (or bad) the last place was.

Too often new people are encouraged to "hit the ground running" when they start a new job – especially if they are in a contract position. To do this denies employees the time to learn and to jettison some of the past and make a fresh start.

I've been guilty of this too, my blood starts the boil the moment I'm introduced to a "project manager", all these assumptions kick in: all they care about is GANTT charts, they believe estimates and the waterfall model, they want to divide, rule and micro-manage. I have to fight these assumptions, ask myself "What proof is there that this project manager is like this?"

Recognising and changing our assumptions isn't easy. It is especially hard when you try and do it on your own. Even looking at data can be confusing, as we tend to see data that supports our point of view rather than data that refutes it.

Writing in the Financial Times, Carne Ross (2005) described the how the British and American Governments argued at the UN with the French and Russian Governments about the 1991-2003 sanctions against Iraq. The two sides cited the same reports to support their case. Ross suggests that both sides were not guilty of ignoring the contradictory evidence, merely that they failed to see it. The assumptions each side made blinded them to contradictory data, they could read the words but their meaning was lost.

We often need other people to help us see our own assumptions; talking problems through helps us understand them and expose our assumptions. Other people come with their own, possibly different assumptions and we can all help highlight one another's. But, when we are locked in confrontation with others we become defensive, to admit an assumption, let alone change it, would be to give ground.

The problem of incorrect and unspoken assumptions affects all aspects of software development: we think we know what the customer wants, or we think we know what the software design should be, but sometimes we're wrong. The need to unlearn assumptions is particularly apparent when it comes to process and literature.

Although it's a great great book I'm getting a bit fed up of people citing Brooks' *Mythical Man Month*. It was written 30 years ago about a project that occurred 40 years ago. Haven't we moved on a bit?

While there is some great advice in Brooks' work there is some we need to unlearn. Lets start with *"Build one to throw away, you will anyway."* Brooks himself has changed his mind on this:

" 'Plan to throw one away; you will anyhow.' This I now perceive to be wrong, not because it is too radical, but because it is too simplistic.

The biggest mistake in the 'Build one to throw away' concept is that it implicitly assumes the classical sequential or waterfall model of software construction."

(Brooks, 1995, p.265)

Then there are *Chief programmer teams*. This is the idea that we can have a few great programmers and arrange things to support them, keep them working productively and all will be right. This approach leads to teams where several lesser programmers work individually on minor pieces of functionality while the *Chief* or *super* programmer(s) delivers the real value. Consciously or unconsciously managers and programmers believe that Jim the super-programmer will deliver 70% of the project while his three helpers will deliver 10% each, of course they'd like four super-programmers but they "can't find them" so they settle for just reducing the load on Jim.

This is quite the opposite of what many people now think and flies in the face of what *Agile* methodologies advocate. Here – as in much of twenty-first century modern business life – it is the team that is important. Whether it is serving fries in McDonalds, building a Nissan or writing software, simply, the scale of modern endeavours means that it is the team that is the building block not the individual.

So, it is with dismay that I hear developers and managers proclaim, "If we only had a few more good people" or "Where can we get more good people?" It is not the lack of individuals that hold our developments back but the lack of good, productive, teams.

We need to apply a bit of unlearning here. Let's try and unlearn this particular mantra.

Sure, maybe one in 100 engineers is more productive than the other 100 put together but are we right to base our entire development process around finding this individual? We need to find them, hire them, motivate them and retain this one person. Even if we can do all that is it right to base an entire strategy around this person? And the chances are, one person isn't enough, we're going to need 10, 20, 100 more like him (and it usually is a him.) And how do these guys work as part of a team? Usually not too well.

It could be that our one individual is actually holding the team back. They may actually block others from dealing with a problem, or their very productivity may hide a fault with the team. Alistair Cockburn tells the following story:

"...a consultant who visited his company and said words to the general effect of, 'If I go into a company and see one super-salesman who is much better than all the rest, I tell the owners to fire this man immediately. Once he is gone, all the others improve and net sales go up'."

(Cockburn, 2003, p.27)

Fact is, the super-programmer approach doesn't scale.

Instead, we need to hire and develop *teams* of people. We give them the tools they need to do the job, and we remove the blockages that stop them from working more productively. We encourage them to improve themselves, their environment, processes and the company – and that means we aren't scared to change, whether it be moving desk or trying a new way of working, we reject the assumption that tomorrow will be a repeat of today.

If we are to expose assumptions we need to enter into open dialogue with others – not necessarily people who hold the same point of view. We need to allow time for this, we need to understand our goals and share mutual goals. Above all we must be prepared to unlearn ourselves, if we start with a position to defend and assumptions we're unwilling to let go of then we aren't going to get very far.

Simple really. Well simple to say, unfortunately, it's incredibly hard to do, after our unlearning we need to learn again.

Allan Kelly
allan@allankelly.net

References

Brooks, F. 1995 *The mythical man month: essays on software engineering*, Addison-Wesley.

Cockburn, A., 2003 *People and Methodologies in Software Development*. PhD. thesis, Oslo

Ross, C., 2005, War Stories, *Financial Times Weekend Magazine*, 29 January 2005

Copy Deadlines

All articles intended for publication in *Overload 67* should be submitted to the editor by May 1st 2005, and for *Overload 68* by July 1st 2005.

Sheep Farming for Software Development Managers
(or Feeling a Bit Sheepish)

by Pippa Hennessy

"Software products are nothing like sheep – they're not soft and cuddly" *Pip's Mother*

It is my contention (and following discussions on the subject with my colleagues I use that word advisedly) that the software development process can be compared to sheep farming. You may find this a little hard to swallow at first (unlike the software as pasta metaphor so ably expounded by Pete Goodliffe [1], which is much more edible), but bear with me and I shall explain all.

My mother's initial reaction (see above) was typical, but the more I thought about it and the more I talked the idea through with others, the more I came to the conclusion that all development processes and the products that emerge from those processes have features in common. I started this exercise more as a joke than anything serious, after all, software products really aren't anything like sheep. However, I've found that many of the techniques, methods, and even truisms that are applicable to software development have very strong parallels in sheep farming – a development process that's been refined over millennia.

Setting the Scene

Last summer I had a job interview. The position I'd applied for was effectively an internal promotion for me, although there were a few external candidates as well. The second interview involved the usual HR-generated "criteria-based" questions, whatever that means, but an equally important part of the interview was for me to give a 30-minute presentation on what I considered the best approach to managing software development within my department. Fair enough, I was applying for the Software Development Manager job.

Now, I've pretty much got the hang of answering interview questions, I can do aptitude tests standing on my head, I always seem to confuse HR people who try to analyse my personality tests (this article may go some way to explain that), and I've even had a job offer resulting from a horrendous interview where they stood me in front of a whiteboard and made me write and explain a C++ program to some highly intelligent nerds. I've also done lots of presentations, ranging from giving papers at conferences to demonstrating prototype software to clients. But for a couple of reasons this particular presentation wasn't anywhere near as straightforward as it should have been. Firstly, I didn't quite know how to pitch it – should I go for "blue sky" and assume no knowledge of the workings of our department, or should I address the problems and issues that I knew (and they knew I knew) existed? Secondly, the interview was two days before the final copy deadline for the ACCU journals (I was the production editor at the time), so my preparation time was drastically reduced, and I didn't even get time to sit down and think about it in any depth until the day before.

So, perhaps understandably, I was a little worried about this presentation. I'd managed to jot down a few ideas in spare

Letter to the Editor

Re: Overload, February 2005 editorial

Hi Alan,

I just want to thank you for your great editorial in the February issue of Overload ("'They' Have Their Reasons"). As you write: *"If you can find and talk to them you will find that "they" are normal human beings trying to achieve reasonable goals in reasonable ways."* Deep inside, I know this to be true, but it's still good to be reminded of it, especially if you deal with someone who appears "unreasonable". They are probably reasonable, from *their* perspective.

What inspired me to write this is that I've lately been discussing with some people in the PHP community, and "they" don't appear to appreciate the value of higher abstractions (preferring a "simple language", even though this may lead to complex or verbose code). Or even stronger type-checking, I may add. In C/C++/Java, we may write:

```
void f(int a, string b, vector c)
```

and we know it takes an `int`, `string` and `vector`, no more, nor less, and returns nothing. In PHP, with the following:

```
function f($a, $b, $c)
```

we know hardly anything: We know that it takes (at least) three arguments – but it may take more – and we know nothing about their type, and any return type (and it may return different types, or nothing at all, depending on its run-time path through the function... All variables in PHP are like variants – they can take on any type). I can't for the life of me understand how someone finds writing code like this more "productive" and "easier". Myself, I've spent way too much time chasing stupid type-related bugs in PHP, that could have trivially been checked by the compiler/runtime. Flexibility to do what? Make type-related errors? Sorry, if I really want to do that, I want to say so explicitly – by overriding the type-system with a cast.

My company makes web applications for other companies, and we're using PHP, and I've been using it professionally for a couple of years, so it's not like I'm a PHP beginner, but still the above baffles me. I like not having to wait for compilation with PHP, but if I could choose stronger type-checking, my answer would be YES. The kind of absent type-checking above tends to encourage sloppy coding, and you have to explicitly check for types inside the function, if you want to do so (it's similar to having to explicitly check for return codes, which is often not done... At least PHP 5 does have exceptions). You still can't enforce a certain return type, though. Really, if I wanted a particular parameter to be a "variant", I'd specify that (and if the types it could have were known, I'd enumerate them, and if not, use something like `boost::any`). It adds information and explicitness to the code, and should make it easier to understand. Yet, it appears the majority of PHP developers don't want it (from what I've read on various mailing lists and newsgroups). Why? Have you got any idea?

Regards,

Terje
<tslettebo@broadpark.no>

moments, when I wasn't busy eating or sleeping or breathing, but with two days to go before the interview I still didn't have much idea of what on earth I'd talk about. I was in a dither as to whether to use my employer's standard Powerpoint template, or whether to even mention that I actually already work with the team I'd hopefully be managing. After a long evening of typesetting Overload I fell into bed and lay awake for a while trying (unsuccessfully) to marshal my ideas into something resembling a coherent framework. And so to sleep. An hour or so later I woke up with a brilliant idea. You know how it is, you think about something, and think, and think, and get absolutely nowhere, and then you wake up in the middle of the night with the Ultimate Solution. Then you go back to sleep again and forget all about it. I was determined that this wasn't going to happen this time, so I lay there in the dark for an hour or two and got it all straight in my head.

What was this idea? Why, that software products are like sheep, and the software development process is just like sheep farming. I have no idea where this came from. Possibly it was a subconscious association with my boss's surname (same as the old bloke who has a farm and sings E-I-E-I-O), or it may be related to my obsession with Lundy (a large lump of granite in the Bristol Channel that's liberally scattered with various kinds of sheep), but it's more likely something to do with my mother, who has a flock of Grey-Faced Dartmoor – a rare and shaggy breed of sheep. Blame the parents, at least that's what I always tell my kids.

Right, enough bleating. It's time to see if I can convince you that software products are indeed soft and cuddly, just like sheep.

Breeds of Software Products

One of the distinguishing features of software products is that they don't actually have many distinguishing features – there are big ones and little ones, simple ones and complex ones, huge monolithic desktop products and straggly internet applications. This is in fact not so different from sheep, of which there are many breeds, each with their own personality. Some have horns, some don't. Some have short wiry hair, some have long shaggy wool. Some are large, some small, some white and fluffy, some brown and scruffy. Some bounce around and play, others simply sit and chew the cud. And as any shepherd will tell you, each individual sheep has its own foibles and eccentricities.

The Ram

Rams don't actually do very much. They're usually bigger than the ewes, and assuming they get past the stage of being lambs (when most rams are slaughtered for meat) their one purpose in life is to father a new generation – for three or four weeks in the autumn they have a great time. So they can be compared to a typical large function-rich desktop software application, whose main purpose is to process and pass on data to other applications and to users. You don't have to do very much with them from year to year, just make sure they're fit for their information-providing purpose.

On a fairly regular basis rams are "rented out" to service other flocks, ensuring widening of the gene pool. This may involve some logistics, such as timing and transport, but in general it's a fairly straightforward process. Similarly, your large data-providing application may be used to process data from other sources, or provide data for a purpose outside its original business specification. This may involve some additional plugins or external data processing to deal with unfamiliar formats, but this tends to be standard operational procedure and as such is relatively trivial.

Incidentally, did you know that when a ram is put with a flock of ewes it has a sachet of dye tied to its midriff so the shepherds can tell which ewes have been serviced by the bright splodges on their backs? This may be stretching the metaphor slightly, but I guess this could be considered similar to data copyright notices that have to be included in reports that are produced using the data.

What can we learn?

Rams are for life, not just for Christmas. Once they're fully grown, they need to be healthy enough to do their job and pull in the stud fees, year in year out, without too much looking after. So it pays to put the effort in to get them right in the first place.

The Old Ewe

This old lady is quite similar to the ram, in that all she really does is to produce one or two healthy lambs every year, thus making a valuable contribution to the productivity of the flock. Again, she's like one of those big desktop products we all know and love, bringing in a steady income without too much attention. She'll probably need more looking after than the ram, but in return she'll provide spin-off products that will in turn go on to contribute to the farmer's income.

What can we learn?

Pretty much the same as the ram. Put the time and resources in up front, and your flock will increase in size without much further input from you.

The Prize Sheep

Farms are quite similar to IT companies in that they need to have a good name among other farmers and farm product consumers. There are a few ways in which a flock can make a name for itself. Lundy has a very successful business selling lamb via the internet – I think this has probably taken off because people who visit Lundy try the lamb while they're on the island and know it's extremely high quality (which, I guess, is another valid comparison – if you see how well a software product performs by trying an evaluation copy you're more likely to buy that product or others from the same company).

Anyway, back to the point. If one or two lambs show all the required qualities of their particular breed they may well be kept for entry into agricultural shows – the ram escapes the chop (and if he's lucky he might be kept for breeding purposes) and the ewe doesn't necessarily have to produce healthy lambs year in year out to be assured of a long and happy life. Rosettes mean the flock is marked as worth considering as a source for butchers or other flocks. How many IT companies can you think of that have based their reputation and their success on one or two flagship products?

What can we learn?

Appearances matter too. Sometimes it's not important for a sheep to bring in much actual revenue – if it is covered in rosettes it will draw attention to the quality of the rest of your flock.

Wild Sheep

There are many flocks of wild sheep in the UK and probably worldwide. These are generally left to look after themselves, possibly with a little judicious tinkering where required. They tend to be hard to control as they're not used to interference by humans or collies, and they often go off in unexpected directions. This is not necessarily a bad thing, as they're hardy and tough and can deal with whatever nature throws at them. Lambing just happens, no need for a shepherd to be up at all hours ready to lend a hand or bottle-feed rejected lambs.

Open source software seems to run along these lines. Applications are developed with an initial idea in mind, and assuming they survive the birthing process and don't fall off a cliff they take on a life of their own, being added to and modified to deal with slightly different problem domains. I guess you could argue that this is due to a high level of external influence, but I imagine that to the original programmer it would seem that their lamb has a life of its own and has taken control of its own destiny.

What can we learn?

Don't ignore a wild sheep because it has no pedigree – it may have evolved attributes that are needed to deal with environments common to all sheep (or a significant subset).

Rare Breeds

The sheep farming community has its equivalent of nerds – those (like my mother) who keep flocks of rare breeds and are actively involved in ensuring these endangered species don't die out. Rare breed shepherds are often also very active in keeping knowledge and use of "outdated" breeding and farming techniques alive along with the sheep. In general these breeds and techniques were once commonplace, but have been superceded by more commercially viable breeds and farming methods.

Rare breeders can be compared to programmers who write ZX Spectrum emulators for XBoxes, or Amstrad PCW emulators for PCs. Often the reason for writing these emulators is to play fondly-remembered games even after the original hardware has fallen to pieces, but a more respectable excuse is to keep old tools and techniques alive just in case they should prove useful in the future. My mother claims this holds true for rare breeds as well – keep the good old standbys around just in case commercial breeding programmes could find a use for the particular traits embodied in breeds that have been around for generations. I'm prepared to give her the benefit of the doubt on this one, especially as I still enjoy playing Hungry Horace on my Psion.

What can we learn?

Don't write off "old" breeds or farming methods, they are tried and tested and can even be better in some situations than their modern equivalents.

Lambs

Lambs are fun. Anyone who tries to tell my mother they're not gets a Hard Stare, and I'm sure many of you secretly number gambolling lambs among the first signs of spring. Newly-born software products are also fun – ask any programmer. You get to try out new ideas, see how high you can make the product jump, see how well it plays with other products, watch it grow – and don't they grow quickly? Coding a new product is like bottlefeeding a lamb – you hang on as tight as you can to the bottle while it gulps the milk down, little tail waggling away madly, then when the bottle's empty, off it goes to skip around and see what mischief it can get into next.

Of course, there is the mortality rate to consider. Mum's flock usually produces around 30 lambs every year, and between one and four of those lambs will not survive. There are those who are stillborn (the sales manager's bright idea for a program that he knows he can sell, which is technically impossible to put into practice), those who die soon after birth (write a prototype, discover the client didn't want it after all, or it's impossible to meet the business requirements), and those who require so much effort and money to keep going that the decision has to be made to let them die. There's a useful shepherd's adage that applies equally to young software products – if a lamb's no good, hit it on the head.

What can we learn?

Let your shepherds and sheepdogs play with the lambs if they want (within reason), it keeps them happy and productive. And be realistic – not all lambs will survive to become useful members of your flock.

Tending the Flock

Right, hopefully by this point I've persuaded you that the metaphor is actually worth exploring. Hey, I've even almost persuaded myself! Anyway, the above is an expansion of the ideas that came to me in the middle of that fateful night. I didn't even start to consider the peripheral issues until later. OK, so sheep are intrinsically like software products. What about their respective development processes?

Stockmen

In general, a flock of sheep is managed by one shepherd on behalf of the farmer. There may be other humans who help manage the flock on a regular basis – farm hands, additional shepherds, the farmer's family. There will certainly be specialists who are called in occasionally (either regularly or in response to specific situations) – vets, shearers, seasonal workers. Many shepherds also keep dogs to help manage the flock – my mother usually enlists the help of my sister, or me and my kids if we're around – which is more or less the same thing. In addition to the farmer, there may be other humans who provide overall direction to the shepherd. There may be a farm shop, whose manager might ask for more lamb, more mutton, more wool, or even more ewe's milk, depending on customer requirements.

There are many (possibly unflattering) parallels to be drawn with software development here. If sheep are the software products, the farmer is the board of directors or CEO of the company, the head shepherd is the software development manager, and the other farm employees (human and canine) are the programmers. Seasonal workers such as shearers would be contractors, and vets and other experts are consultants. The chap who runs the farm shop may be an account manager, product manager, or possibly the marketing manager.

Interestingly, shepherds are as reluctant to spend money on outside experts as software development managers. If there's a

problem, every possible home remedy (often passed down from father/mother to daughter/son through the ages) will be tried first before calling in the vet. This is usually OK – these folklore cures have been tried and tested for centuries after all – but it can be counterproductive in the extreme if modern veterinary science has come up with a cheap simple solution to the problem. The mark of a good shepherd is one who keeps up with the times and knows when to call in the experts – qualities also required for managing software development.

What can we learn?

Shepherds do not work alone. Being an expert at dealing with sheep is not enough, a shepherd has to be able to communicate with other humans and canines to get the job done. S/he also has to be able to take in and effectively process and use information relevant to his/her work, adapting to specific and general changes in situations.

Breeding a Flock

There are many ways to develop a flock of sheep. You start with some ewes – possibly just one or two, maybe a whole existing flock, but more likely somewhere in between. You might buy two or three year old ewes that are already proven to be good breeders, or you might buy ewe lambs, who may or may not produce lambs in the years to come. Over the next few years you need to make decisions about which ram (or rams) to use to service your ewes, which lambs to keep and which to sell, whether to buy in new ewes to widen the gene pool, whether to "retire" old ewes that don't produce viable offspring. With the strides made in cloning technology it may even be possible at some point in the future to make copies of your best sheep – although there are currently problems with this process, I believe Dolly the Sheep suffered terribly from arthritis and died at an early age. All the time you have to balance the immediate profitability of your flock against investments for future prosperity – you may know that in five years' time you'll have the best flock in the county if you pay a fortune to have Billy the SuperRam service your ewes this autumn, but that's no use to you if you then can't afford to feed your sheep in the spring.

Each of these options has advantages and disadvantages, which I won't go into here as they're not particularly relevant. What is interesting is that there are also several almost directly comparable ways to develop a portfolio of software products. I'd guess most software development organisations start with one or two ideas that are developed into products. These will inevitably generate offspring – related products which are either add-ons to the original software or modified versions for different markets, or new products for the existing market. As the company expands, it may buy in and take over development of other software products (with all the pain that can entail – no amount of due diligence will expose all the potential time-bombs in someone else's code), and it will certainly produce newer and better versions of much of its existing product set and continue to develop new products. Software that is no longer profitable for whatever reason will be deprecated and eventually no longer supported. As with sheep farming, the bottom line is all-important. Developing a product portfolio is expensive, there must be money coming in from existing products to support the resources required to develop new products.

What can we learn?

Planning is everything. Having a long-term strategy for the flock is essential, but the farmer must also know how he's going to cover the day-to-day expenses while putting that strategy into practice (and if necessary the long-term strategy must be modified if it is not supportable).

Growth and Development

Sheep need grass. They don't actually need too much else – if you lived on a desert island with a flock of sheep and a field of grass you'd probably be OK. Sure, you'd be better off if you had a trusty collie, or access to a vet and modern medications, or bedding and pens, but your flock would survive as long as it could graze. In the real world, if the grass is poor quality you'll need to give your flock supplements – typically salt licks and various trace elements in some form. You may also need to provide more bulk in the form of hay. Conversely if the grass is too rich you get interesting things happening at the other end (the technical term is "daggy sheep") and you'll need to clean their bottoms regularly to avoid all sorts of nasty possibilities.

Software products need a programming language, and not much else. The equivalent of the field is the compiler/linker or interpreter for that language, and the hardware/operating system to run it on. If it's a functionally sparse language you're likely to need to implement elements in other languages or call other services – I vaguely remember that if you were going to do anything worth doing with ZX Spectrum Basic you'd have to write chunks of assembler code. On the other hand, if it's an all-singing all-dancing language like C++, your expert developers will be able to produce fantastically elegant, fast and highly functional programs... but when another (usually junior) developer comes to de-dag the bottom end (in order to fix a bug or extend the software) they've got an awful lot of untangling to do to work out what's going on.

There are many other minor not-quite-requirements that a sheep farmer has to take into account: transporting sheep to market or slaughter (packaging up software for distribution), tagging and numbering sheep so they can be identified (setting up project ID codes and directory structures), obtaining and paying for the services of experts, for example vets and shearers (employing consultants and contractors), providing and maintaining bedding, fences, gates, pens as required (buying and maintaining hardware resources), meeting health and safety regulations for meat produce or meeting ideal standards for shows (QA to ensure software meets user requirements)[1].

What can we learn?

Decisions taken about the basic requirements for the development of your flock will have consequences and potential hidden costs – be aware of these as far as possible and make informed decisions. Also, it's not enough just to breed the perfect sheep, you need to manage its growth and keep it healthy if you intend to make any profit from it.

Bugs

Sheep have bugs. Lots of them. A major part of a shepherd's job is to keep an eye on his/her flock for any signs of disease,

1 Incidentally, a general rule of thumb for the "perfect sheep" is it should be like a table – a flat top with a leg at each corner. If only user requirements for software products were so simple!

and to administer prophylactic treatments regularly to stave off the possibility of infestations or infections. Preventative measures such as spraying the entire flock once or twice a year with insecticide and regularly dosing for worms are taken as a matter of course, although they're not much fun to do. It is second nature for a good shepherd to spot symptoms of acute infections early on so that they can be treated before permanent damage is done to the flock (although see comments on calling in the vet in the Stockmen section above). Shepherds are also very conscious of what's going on with the neighbours – infestations such as lice or viruses can spread very easily between flocks in adjacent fields, and if allowed to proliferate can be potentially devastating – look at the short and long term effects of the last foot and mouth outbreak.

As software developers we too have to deal with bugs. (Thankfully this task isn't quite as offensive or messy as the shepherd's version though.) We all write and run unit tests as a matter of course – don't we? – so we spot and eliminate potentially nasty infections before they can do much damage. We have QA processes and (usually) a dedicated testing team to run detailed and exhaustive checks for bugs, and when they find a bug we deal with it according to its severity and impact. We also install firewalls and virus-checkers on our computers to ensure our virtual neighbours can't pass any nasty surprises on to us, and a good software developer will consider security issues during development of any internet product.

What can we learn?

In general bugs are fairly apparent in both sheep and software products, and must be dealt with as a matter of urgency. It is important to spot bugs as early as possible in the development process – it's far cheaper to treat a single sheep for a maggot infestation than to treat a whole flock.

General Cussedness

Sheep are stupid. If one sheep gets it into its head to run in the wrong direction you can guarantee that most of the rest of them will decide it's a good idea and follow it. And there's a good reason why one of the standard tests at sheepdog trials is to split off a given number of sheep from the flock – sheep are *stupid.* They can also be quite vicious – you might introduce a new ewe to your flock, only to find one of your existing ewes takes a dislike to it and turns on it. I've even seen a ewe push her own lamb into a crevasse to get rid of it.

Software can be stupid and vicious too (well, sort of). If your team has always developed software one way, it is often very difficult to effectively steer the products in a different direction. One example that springs to mind is the development process itself – if you've always used the waterfall method it's remarkably difficult to introduce elements of agile development. Or you might find that one product is developed using extreme programming, and everyone likes it so much that they race off and start using XP for everything, whether or not it's appropriate. Another example might be the technologies used – we've developed this product using ASP and JavaScript, so we've got to develop all our internet products that way, whether or not ASP.NET and web services makes more sense. As for vicious software, I can think of countless examples of projects where it's seemed sensible to save some time by integrating a third party component – it never seems to quite work out that way.

What can we learn?

Don't let your sheep (or your stockmen or collies) be stupid – make decisions on a case by case basis. "We've always done it that way," or "They did it that way with that sheep and it worked brilliantly," are not in themselves good reasons for making decisions. And be very wary of new sheep – they may be incompatible with your existing flock.

Conclusion

Well, what can one possibly conclude from these slightly insane ramblings? Hopefully not that I shouldn't have got the job – I did forbear from expanding on the sheepy metaphor in my presentation, although I couldn't resist mentioning it – my boss was quite keen to hear about it, so I guess I'll have to show him this article and hope he doesn't demote me (or have me sectioned) on the spot.

I could get into a bit of theology here – for example, Christians use the sheep idea a lot. But I don't think I will, there are enough holy wars about code layout and other such important issues in the programming community as it is.

My preferred conclusion is that a shepherd's crook, a couple of well-trained collies, and a quad bike should be required equipment for a software development manager, and should be provided by every enlightened employer.

I guess if I were to be serious for a moment (it does happen occasionally), I'd propose that it's possible to find useful ideas and process elements and rules of thumb in the strangest of places. People have been farming and breeding sheep for thousands of years – the development process that produces sheep that are "fit for purpose" has been refined and improved upon for generations, and although software products really aren't much like sheep the processes of developing those products have sometimes surprising parallels which we can all learn from.

My favourite example has to be, "If a lamb's no good, hit it on the head." A shepherd will do this without a second thought. How many times have you been involved in trying to rescue an infant software product that is obviously too feeble to survive, and wondered why you're bothering?

Pippa Hennessy
pip@oldbat.co.uk

Acknowledgments

Thanks to my mother for many entertaining and informative conversations about the ins and outs of sheep farming, to several of my colleagues for joining me in thinking out of the sheep pen, and to Mum, Phil Bass and Alan Griffiths for reviewing the initial draft.

Any errors in my representation of the ins and outs of sheep farming (and software development, for that matter) are entirely my own.

References

[1] Pete Goodliffe, "Professionalism in Programming #21 - Software Architecture", *C Vu 15.4*, August 2003

Metaprogramming is Your Friend

by Thomas Guest

Introduction

Whenever I create a new C++ file using Emacs a simple elisp script executes. This script:

- places a standard header at the top of the file,
- works out what year it is and adjusts the Copyright notice accordingly,
- generates suitable `#include` guards (for header files),
- inserts placeholders for Doxygen comments.

In short, the script automates some routine housekeeping for me.

Nothing extraordinary is going on here. One program (the elisp script) helps me write another program (the C++ program which needs the new file).

By contrast, C++ template-metaprogramming is extraordinary. It inspires cutting-edge C++ software; it fuels articles, newsgroup postings and books [Abrahams and Gurotovy]; and it may even influence the future direction of the language.

Despite (or maybe because of) this, this article has little more to say about template-metaprogramming. Instead we shall investigate some ordinary metaprograms. For example, the elisp script – a program to write a program – is a metaprogram. There may be other metaprograms out there which, perhaps, we don't notice. And there may be other metaprogramming techniques which, perhaps, we should be aware of.

What is Metaprogramming?

I like the definition found in the [Wikipedia]:

> *"Metaprogramming is the writing of programs that write or manipulate other programs (or themselves) as their data or that do part of the work that is otherwise done at runtime during compile time."*

Actually, it's the first half of this definition I like (everything up to and including "data"). The second seems rather to weaken the concept by being too specific, and in my opinion its presence reflects the current interest in C++ template-metaprogramming – but a Wikipedia is bound to relect what's in fashion!

Why Metaprogram?

Having established what metaprogramming is, the obvious follow-up is "Why?" Writing programs to manipulate ordinary data is challenging enough for most of us, so writing programs to manipulate programs must surely be either crazy or too clever by half.

Rather than attempt to provide a theoretical answer to "Why?" at this point, let's push the question on the stack and discuss some practical applications of metaprogramming.

Editor Metaprogramming

I've already spoken about programming Emacs to create C++ files in a standard format. We can compare this technique to a couple of common alternatives:

1. create an empty file then type in the standard header etc.
2. copy an existing file which does something similar to what we want, then adapt as required.

The first option is tough on the fingers and few of us would fail to introduce a typo or two. The second is better but all too often is executed without due care – maybe because a programmer prefers to concentrate on what she wants to add rather than on what she ought to remove – and all too often leads to a new file which is already slightly broken: perhaps a comment remains which only applies to the original file, perhaps there's an incorrect date stamp.

The elisp solution is an improvement. It addresses the concerns described above and can be tailored to fit our needs most exactly. All decent editors have a macro language, so the technique is portable.

Of course, there is a downside. You have to be comfortable customising your editor. (Or you have to know someone who can do it for you.)

Batch Editing

By "batch editing" I mean the process of creating a program to edit a collection of source files without user intervention. This is closely related to editor metaprogramming – indeed, I often execute simple batch edits without leaving my editor (though the editor itself may shell-out instructions to tools such as `find` and `sed`).

Very early on in my career (we're talking early 80's) I worked with a programmer who preferred to edit source files in batch mode. His desk did not have a computer terminal on it. Instead, he would study printouts, perhaps marking them up in pencil, perhaps using a rubber to undo these edits, before finally writing – by hand – an editor batch file to apply his changes. He then visited a computer terminal to enter and execute this batch file.

Even then, this was an old-fashioned way of working, yet he was clear about its advantages:

- **Recordable:** the batch file provides a perfect record of what it has done.
- **Reversible:** its effects can therefore be undone, if required.
- **Reflective:** by working in this reflective, careful way, he was less likely to introduce errors. When system rebuilds can only be run overnight, this becomes paramount.

These days, builds are quicker and batch editing is more immediate. With a few regular expressions and a script one can alter every file in the system in less time than it takes to check your email. As an example, in another article [Guest1] I describe the development of a simple Python script to relocate source files into a new directory structure, taking care to adjust internal references to `#included` files.

The benefits of using a script to perform this sort of operation are a superset of those listed above. In addition, a scripted solution beats hand hacking since it is:

- **Reliable:** the script can be shown to work by unit tests and by system tests on small data sets. Then it can be left to do its job.
- **Efficient:** editing dozens – perhaps hundreds – of files by hand is error prone and tedious. A script can process megabytes of source in minutes.

Again, there is a downside. You have to invest time in writing the script, which may well require a larger investment in learning a new language. Many of us would regard proficiency in other languages as an upside but it may be difficult to make that initial investment under the usual project pressures.

So, once again, it may end up being a team-mate who ends writes the script for you. Indeed, many software organisations have a dedicated "Tools Group" which specialises in writing and customising tools for internal use during the development of core products. Perhaps this team could equally well be named a "Metaprogramming Group"?

Compilation

The compiler is the canonical example of a metaprogram: it translates a program written in one language (such as C) into an equivalent program written in another language (object code).

Of course, when we invoke a compiler we are not metaprogramming, we are simply using a metaprogram, but it is important to be aware of what's going on. We may prefer to program in higher-level languages but we should remember the compiler's role as our translator.

We lean on compilers: we rely on them to faithfully convert our source code into an executable; we expect different compilers to produce "the same" results on different platforms; and we want them to do all this while tracking language changes.

In some environments these considerations are taken very seriously. For safety critical software, a compiler will be tested systematically to confirm the object code produced from various test cases is correct. In such places, you cannot simply apply the latest patch or tweak optimisation flags. You may even prefer to work in C rather than C++ since C is a smaller language which translates more directly to object code.

In other environments we train ourselves to get along with our compilers. We accept limitations, report defects, find workarounds, upgrade and apply patches. Optimisation settings are fine-tuned. We prefer tried-and-tested and, above all, supported brands. We monitor newsgroups and share our experiences.

One last point before leaving compilers alone: C and C++ provide a hook which allows you to embed assembler code in a source file – that's what the `asm` keyword is for. I guess this too is metaprogramming in a rather back-to-front form. The `asm` keyword instructs the compiler to suspend its normal operation and include your handwritten assembler code directly. Its exact operation is implementation dependent, and, fortunately, rarely needed.

Scripting

The program which follows is a short but non-trivial Python script. It makes use of a couple of text codecs from the Python standard library to generate a C++ function. This C++ function converts a single character from ISO 8859-9 encoding into UTF-8 encoded Unicode.

```
def warnGenerated():
  '''Return a standard 'generated code' warning.'''
  import sys, time
  return (
    '// GENERATED CODE. DO NOT EDIT!\n'
    '// generated by %s, %s' %
    (' '.join(sys.argv),
     time.asctime(time.localtime()))
    )

def functionHeader(codec):
  '''Return the decode function header.'''
  return '''/**
* @brief Convert from %(codec)s into UTF-8
* encoded Unicode
* @param %(codec)s An %(codec)s encoded character
* @param it Reference to an output iterator
* @note If the input character is invalid, the
* Unicode replacement character U+FFFD will be
* returned.
*/
template <typename output_iterator>
void
%(codec)s_to_utf8(
  unsigned char %(codec)s,
  output_iterator & it)''' % { 'codec' : codec }

def convertCh(ch, codec):
  '''Return the 'case' statement converting
  the input character using the supplied codec'''

  from unicodedata import name

  ucs = chr(ch).decode(codec, 'replace')
  utf = ucs.encode('utf-8')
  ucname = name(ucs, 'Control code')
  action = '; '.join(['*it++ = 0x%02x' % ord(c)
                      for c in utf])

  return '''case 0x%02x: // %s
  %s;
  break;''' % (ch, ucname, action)

def codeBlock(prefix, body, indent = ' ' * 4):
  '''Return an indented code block.
  This code block will be formatted:
  prefix
  {
    body
  }'''
  import re
  indent_re = re.compile('^', re.MULTILINE)
  return '''%s
{
%s
}''' % (prefix, indent_re.sub(indent, body))

codec = 'iso8859_9'
print warnGenerated()

print codeBlock(
  functionHeader(codec),
  codeBlock(
    'switch(%s)' % codec,
    # iso8859-* encodings are 8-bit
    '\n'.join([convertCh(ch, codec)
               for ch in range(0x100)]),
    indent = '' # don't indent case: labels
    )
  )
```

By now, it should go without saying that this script is a metaprogram. Before discussing why I think it's a good use of metaprogramming, some notes:

- The function `warnGenerated()` is used to place a standard warning in front of the generated C++ function. If users of this C++ function edit it by hand, their changes will be overwritten next time the script is run: hence the warning.
- The generated code identifies the command which created it (this information appears as part of the standard warning). This is to help users regenerate the code, if required.

- It is very important that the Python script is both maintained and easy to locate. Ideally, the build system includes a rule to generate the C++ from the script, though this behaviour may be hard to integrate into some IDEs: it may prove more pragmatic to run the script by hand and keep the dependent C++ code checked directly into the repository.
- Notice how Python's triple quoted strings allow us to create neatly formatted C++ code from neatly formatted Python code without needing lots of escaped characters.
- It is perhaps ironic that, according to the Python documentation, some of Python's builtin codecs are implemented in C (presumably for reasons of speed). I haven't worked out if this applies to the ones this script uses.

I like this script since it makes use of the standard Python library to create code we can use in a C++ program. The hard work goes on in the calls to `encode()` and `decode()` and we don't even have to look at the implementations of these functions, let alone maintain them. Their speed does not affect the speed of our C++ function and I am willing to trust their correctness, meaning I don't have to locate or purchase the ISO 8859 standards.

The second big win is that all the boilerplate code is generated without effort. If, at some point in the future, we need a fuller range of ISO 8859 text converters, then we tweak the script so the final section reads, for example:

```
codecs = ['iso8859_%d' for n in range(1, 10)]

print warnGenerated()

for codec in codecs:
  print codeBlock(
    functionHeader(codec)
    ....
    )
```

and let it run. And should we decide on a different strategy for handling invalid input data, again, the metaprogram is our friend.

Preprocessor Metaprogramming

As mentioned in passing, C++ has a sophisticated templating facility which (amongst other things) makes metaprogramming possible without needing to step outside the language.

C++ also inherits the C preprocessor: a rather unsophisticated facility, but one which is equally ready for use by metaprogrammers. In fact, careful use of this preprocessor can allow you to create generic C algorithms and simulate lambda functions.

For example:

```
#define ALL_ITEMS_IN_LIST(T, first, item, ...) \
do {                                           \
   T * item = first;                           \
   while (item != NULL) {                      \
     __VA_ARGS__;                              \
     item = item->next;                        \
   }                                           \
} while(0)

#define ALL_FISH_IN_SEA(first_fish, ...) \
        ALL_ITEMS_IN_LIST(Fish, first_fish, \
                          fish, __VA_ARGS__)
```

The first macro, `ALL_ITEMS_IN_LIST`, iterates through items in a linked list and optionally performs some action on each of them. It requires that list nodes are connected by a next pointer called `next`. The second macro, `ALL_FISH_IN_SEA`, specialises the first: the node type is set to `Fish *` and the list node iterator is called `fish` instead of `item`.

Here's an example of how we might use it:

```
/**
 * @brief Find Nemos
 * @param fishes Linked list of fish
 * @returns The number of fish in the list called
 * Nemo
 */
int findNemo(Fish * fishes) {
  int count;

  ALL_FISH_IN_SEA(fishes,
     if(!strcmp(fish->name, "Nemo")) {
       printf("Found one!\n");
       ++count;
     }
  );
  return count;
}
```

Note how simple it is to plug a code snippet into our generic looping construct. I have used one of C99's variadic macros to do this (these are not yet part of standard C++, but some compilers may support them).

I hesitate to recommend using the preprocessor in this way for all the usual reasons [Sutter]. That said:

- This is a technique I have seen used to good effect in production code.
- Techniques like these are used in highly respected C software – Perl and Zlib, for example. All C/C++ programmers should be familiar with it.
- Although the preprocessor can be dangerous, the way it operates is simple and transparent: use your compiler's `-E` option (or equivalent) to see exactly what the preprocessor is up to. (I sometimes wish I had an equivalent option for working out how the compiler is handling templated code)
- Template metaprogramming experts use every preprocessor trick in the book. See, for example, some of Andrei Alexandrescu's publications [Alexandrescu], or the Boost preprocessor library [Boost]. (This library's documentation includes an excellent introduction to the preprocessor's limitations, and techniques for working round them.)

One final point: the `inline` keyword (intentionally) does not require the compiler to inline code. The preprocessor can do nothing but!

Reflection and Introspection

Take a look at the following Python function which on my machine lives in `<PYTHONROOT>/Lib/pickle.py`

```
def encode_long(x):
  r"""Encode a long to a two's complement
  little-endian binary string.
  Note that 0L is a special case, returning
```

```
an empty string, to save a byte in the
LONG1 pickling context.
>>> encode_long(0L)
''
>>> encode_long(255L)
'\xff\x00'
>>> encode_long(32767L)
'\xff\x7f'
>>> encode_long(-256L)
'\x00\xff'
>>> encode_long(-32768L)
'\x00\x80'
>>> encode_long(-128L)
'\x80'
>>> encode_long(127L)
'\x7f'
>>>
"""
....
```

The triple quoted string which follows the function declaration is the function's docstring (and the `r` which prefixes the string makes this a raw string, ensuring that the backslashes which follow are not used as escape characters). This particular docstring provides a concise description of what the function does, fleshed out with some examples of the function in action. These examples exercise special cases and boundary cases, rather like a unit test might.

Python's `doctest` module [Doctest] enables a user to test that these examples work correctly. Here's how to `doctest pickle` in an interactive Python session:

```
>>> import pickle
>>> import doctest
>>> doctest.testmod(pickle)
(0, 14)
```

The test result, `(0, 14)`, indicates 14 tests have run with 0 failures. For more details try `doctest.testmod(pickle, verbose=True)`. In case anyone is confused, 7 of the tests apply to `encode_long` – and unsurprisingly the other 7 apply to `decode_long`.

Incidentally, if `pickle.py` is executed (rather than imported as a library) it runs these tests directly.

The `doctest` module is a metaprogram – an example of Python being used to both read and execute Python. To see how it works I suggest taking a look at its implementation. The code runs to about 1500 lines of which the majority are documentation and many of the rest are to do with providing flexibility for more advanced use.

In essence, note that docstrings are not comments, they are formal object attributes. Now, Python allows you to list and categorise objects at runtime, so we can collect up the docstrings for classes, class methods and for the module itself. Once we have all these docstrings we can search them to find anything which looks like the output of an interactive session using Python's text parsing capabilities. The remaining twist is Python's ability to dynamically compile and execute source code using the `compile` and `exec` commands. So, we can replay the documentation examples, capturing and checking the output.

The `doctest` module provides no more than an introduction to metaprogramming in Python. Given a Python object it is possible to get at the object's class, which is itself an object which can be dynamically queried and even modified at run-time. This isn't the sort of trick which is often required: I haven't tried it myself so I'd better keep quiet and refer you to the experts. See for example [van Rossum] or [Raymond].

Domain Specific Extensions

Sometimes the best way to solve a particular family of problems is to create a domain specific language, which may be implemented as an extension to a standard language

For example (and once again, quite early in my career), I worked for an organisation – I'll call it Vector Products – which specialised in solid geometry software. Vector Products developed and actively maintained a proprietary extension to C – I'll call it C-cubed – which provided native support for various domain-specific primitives: vectors (the sort you find in 3D mathmematics, not `std::vector`), ranges, axis-aligned boxes; and for domain specific operators to work with these primitives.

I should stress that this C extension pre-dated standard C++. C++ classes and operator overloading can now handle much of what C-cubed provided. Nonetheless, Vector Products' investment paid off: C-cubed allowed programmers to write vector mathematics in a clean and legible way, thereby freeing them to concentrate on the real solid geometry problems they needed to solve.

I believe that the earliest incarnations of C++ were essentially domain-specific extensions to C. For early C++, the domain would be "Object Oriented Programming". [Stroustrup]

This again is metaprogramming, though (particularly with respect to the supplied examples) it is closely related to compilation.

Metaproblems

Most of this article puts a positive spin on metaprogramming. I'm happy enough to leave you with this impression, but I should also mention some problems.

Trouble-shooting

The first problem is to do with trouble-shooting. You have problems with your program but the problem is actually in the metaprogram which generated your program. You are one step removed from fixing it.

I deliberately used the term "trouble-shooting" rather than debugging. When you think about it, debug builds and debuggers are there to help you solve these problems by hooking you back from machine code to source code. It gives the illusion of reversing the effect of the compiler. If you can provide similar hooks in your metaprograms, then similarly the fix will be easier to find.

Quote Escape Problems

The second problem I refer to as the "quote-escape" problem. It bit me recently when I converted a regular C++ program into one which was partially generated by another C++ program. For details, I refer you to [Guest2].

For the purposes of this article, look at what happened when I needed to generate C++ code which produces formatted output.

Here's the code I wanted to generate:

```
context.decodeOut()
  << context.indent()
  << field_name << " "
  << bitwidth
  << " = 0x" << value << "\n";
```

Here's the code I developed to do the generating:

```
cpp_file
  << indent()
  << "context.decodeOut() << context.indent() << "
  << quote(field_name
        + " "
        + bitwidth
        + " = 0x")
  << " << context.readFieldValue("
  << quote(field_name) + ", "
  << value
  << ") << \"\\n\";\n";
```

It looks even worse without the helper function, `quote`, which returns a double-quoted version of the input string.

I was able to defuse this problem with some refactoring but the self-referential nature of metaprogramming will always make it susceptible to these issues.

This is also part of the reason why Python is so popular as a code-generator: as has been shown by some of the preceding examples, its sophisticated string support can subvert most quote-escape problems.

Build Time Complexity

I have already mentioned the problem of integrating code-generators into your build system. Some IDEs don't integrate them very well, and even if they do, we have introduced complexity into this part of the system. In general we prefer to trade complexity at build time for safety at run-time but we should always check that the gains outweigh the costs.

Too Much Code

We're nearing the end of our investigation, and I hope the "Why Metaprogram?" question I posed at the beginning has been addressed. The [Wikipedia] answers this question rather more directly:

> *"[Metaprogramming] ... allows programmers to produce a larger amount of code and get more done in the same amount of time as they would take to write all the code manually."*

It's possible to interpret this wrongly. As we all know, we want less code, not more (more software can be good, though). The important point is that the metaprogram is what we develop and maintain and the metaprogram is small: we shouldn't have to worry about the generated code's size.

Unfortunately we do have to worry about the generated code, not least because it has to fit in our system. If we turn a critical eye on the ISO 8859 conversion functions we discussed earlier we can see that the generated code size could be halved: values in the range (`0`, `0x7f`) translate unchanged into UTF-8, and therefore do not require 128 separate cases. Of course, the metaprogram could easily be modified to take advantage of this information, but the point still holds: generated code can be bloated.

See [Brown] for a more thorough discussion of this issue.

Too Clever

Good programmers use metaprograms because they are lazy. I don't mean lazy in the sense of "can't be bothered to put the right header in a source file", I mean lazy in the sense of "why should I do something a machine could do for me?".

Being lazy in this way requires a certain amount of cleverness and "clever" can be a pejorative every bit as much as "lazy" can. A metaprogram lives at a higher conceptual level than a regular program. It has to be clever.

Experienced C++ programmers are used to selecting the right language features for a particular job. Where possible, simple solutions are preferred: not every class needs to derive from an interface, and not every function needs template-type parameters. Similarly, experienced metaprogrammers do not write metaprograms when they can, they do it when they choose to.

Concluding Thoughts

This article has touched on metaprogramming in a few of its more common guises. I hope I have persuaded you that metaprogramming is both ubiquitous and useful, and that it shouldn't be left to a select few.

At one time, the aim of computer science seemed to be to come up with a language whose concepts were pitched at such a high level that software development would be simple. Simple enough that people could program machines as easily as they could, say, send a text message[1]. Compilers would be intelligent and forgiving enough to translate wishes to machine code.

This aim is far from being realised. We do have higher-level languages but their grammars remain decidedly mechanical. Programs written in low-level languages still perform the bulk of processing. Perhaps a more realistic aim is for a framework where languages and programs are compatible, able to communicate with humans and amongst themselves, on a single device or across a network.

In such a framework, metaprogramming is your friend.

Thomas Guest
thomas.guest@ntlworld.com

References

[Abrahams and Gurtovoy] David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*

[Alexandrescu] Andrei Alexandrescu's homepage is at `http://www.moderncppdesign.com/main.html`

[Brown] Silas S Brown, "Automatically-Generated Nightmares", *CVu 16.6*

[Doctest] doctest – Test interactive Python examples `http://docs.python.org/lib/module-doctest.html`

[Guest1] Thomas Guest, "A Python Script to Relocate Source Trees", *CVu 16.2* (also available re-titled "From A to B with Python" at [Homepage]

[Guest2] Thomas Guest, A Mini-Project to Decode a Mini-Language - Part 3, available at [Homepage]. (The first two parts of this article appeared in Overloads 63 and 64).

[concluded at foot of next page]

1 Though maybe we aren't so far off. To quote Bjarne Stroustrup [Stroustrup2]: *"I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone."*

Separating Interface and Implementation in C++

by Alan Griffiths & Mark Radford

This article discusses three related problems in the design of C++ classes and surveys five of the solutions to them found in the literature. These problems and solutions are considered together because they relate to separating the design choices that are manifested in the interface from those that are made in implementing the class. The problems are:

- Reducing implementation detail exposed to the user
- Reducing physical coupling
- Allowing customised implementations

These have led developers to seek ways to separate interface from implementation and practice has seen all of the following idioms used and documented. We will be evaluating them to see how they compare as solutions to the above problems:

- Interface Class
- Cheshire Cat
- Delegation
- Envelope/Letter
- Non-Virtual Public Interface

In order to illustrate the problems and solutions we are going to use a telephone address book (with very limited functionality) as an example. For comparison purposes we have implemented this as a naïve implementation (see first sidebar) which does not attempt to address any of the stated problems. We have also refactored this example to use each of the idioms – the header files are reproduced in the corresponding sidebars. (The full implementation and sample client code for all versions of the example are available with the online version of this article [WEB05].)

Examining the Problems

Problem 1: Reducing Implementation Detail Exposed to the User

Client code makes use of an object via its public interface, without any recourse to implementation details. Since the authors of client code have to use an object through its public interface that interface is all they need to understand. This public interface typically comprises member function declarations.

C++ allows developers to separate the implementation code for member functions from the class definition, but there is no comparable support for separating the member data that implements an object's state (or, for that matter, for separating the declarations of private member functions). Consequently the implementation detail exposed in a class's definition is still there as background noise, providing users with an added distraction. The definition of a class is typically encumbered with implementation "noise" that

Naïve Implementation

```
// naive.h – implementation hiding example.
#ifndef INCLUDED_NAIVE_H
#define INCLUDED_NAIVE_H
#include <string>
#include <utility>
#include <map>

namespace naive {
/** Telephone list. Example of implementing a
*     telephone list using a naive implementation.
*/
class telephone_list {
public:
  /** Create a telephone list.
  * @param    name    The name of the list.
  */
  telephone_list(const std::string& name);

  /** Get the list's name.
  * @return   the list's name.
  */
  std::string get_name() const;

  /** Get a person's phone number.
  * @param    person   Person's name (exact match)
  * @return   pair of success flag and (if success)
  *             number.
  */
  std::pair<bool, std::string>
  get_number(const std::string& person) const;

  /** Add an entry. If an entry already exists for
  *     this person it is overwritten.
  *   @param  name    The person's name
  *   @param  number  The person's number
  */
  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  typedef std::map<std::string, std::string> dictionary_t;
  std::string  name;
  dictionary_t dictionary;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(const telephone_list& rhs);
};
} // namespace naive
#endif
```

[continued from previous page]

[Homepage] http://homepage.ntlworld.com/thomas.guest
[Raymond] Eric S. Raymond, *Why Python?*, http://pythonology.org/success&story=esr
[Stroustrup1] Bjarne Stroustrup, *The Design and Evolution of C++*
[Stroustrup2] Bjarne Stroustrup, *Did you really say that?*, from Bjarne Stroustrup's FAQ http://www.research.att.com/~bs/bs_faq.html#really-say-that
[Sutter] Herb Sutter, "What can and can't macros do?", *Guru of the Week 77* http://www.gotw.ca/gotw/077.htm
[van Rossum] *Unifying types and classes in Python 2.2* http://www.python.org/2.2/descrintro.html
[Wikipedia] A free-content encyclopedia that anyone can edit, http://wikipedia.org/

Credits

Thanks to Dan Tallis for reviewing an earlier draft of this article.

is of no interest to the user and is inaccessible to the client code written by that user: the naïve implementation shows this with `MyDict`, `myName` and `dict`.

Problem 2: Reducing Physical Coupling

The purpose of defining a class in a header file is for the definition of that class to be included in any translation units that define the client code for that class. If classes are designed in a naïve manner this leads to compilation dependencies upon details of the implementation that are not only inaccessible to the client code but also (in most cases) do not affect it in any way.

These compilation dependencies are undesirable for two reasons:

- Additional header file inclusions may be required to compile the class definition. This increases the size of all dependent translation units. The "Naïve Implementation" example needs `<map>` even though `std::map` is not used in the public interface – if this were a user header with its own inclusions these too might be "bloat".
- When changes are made to implementation elements in the header – even without affecting the interface – the client code must be recompiled. (When using shared libraries this can also introduce binary incompatibilities between versions.) Should the example implementation change the choice of using `MyDict`, `myName` or `dict` this affects all client code.

In a medium to large system the effect of these compilation dependencies can multiply to an extent that causes excessive and problematic build times.

Problem 3: Allowing Customised Implementations

Library code frequently defines points of customisation for user code to exploit. One of the ways to do this is to specify an interface as a class and allow the user code to supply objects that conform to this interface.

Such a library is typically compiled before the user code is written. In this case the library contains the "client code" and for this to have compilation dependencies on the implementation would be problematic.

Clearly, the naïve implementation makes no provision for alternative implementations.

The Idioms

We present the best known idioms for implementation hiding along with some comments in italics.

Each of these idioms can have advantages and these need to be understood when choosing between them.

Cheshire Cat

A private "representation" class is written that embodies the same functionality and interface as the naïve class – however, unlike the naïve version, this is defined and implemented entirely within the implementation file. The public interface of the class published in the header is unchanged, but the private implementation details are reduced to a single member variable that points to an instance of the "representation" class, each of its member functions forwards to the corresponding function of the "representation" class.

The term "Cheshire Cat" (see [Murray1993]) is an old one, coined by John Carollan over a decade ago. Sadly it seems to have disappeared from use in contemporary C++ literature. It appears described as a special case of the BRIDGE pattern in "Design Patterns" [GOF95], but the name "Cheshire Cat" is not mentioned. Herb Sutter (in [Sut00]) discusses it under the name "Pimpl idiom", but considers it only from the perspective if its use in reducing physical dependencies. It has also been called "Compilation Firewall".

Cheshire Cat requires "boilerplate" code in the form of forwarding functions (see "Cheshire Cat Implementation" sidebar below) that are tedious to write and (if the compiler fails to optimise them away) can introduce a slight performance hit. It also requires care with the copy semantics (although it is possible to factor this out into a smart pointer – see Griffiths99). As the relationship between the public and implementation classes is not explicit it can cause maintenance issues.

Delegation

One or more areas of the class functionality are factored out from the naïve implementation into separate helper classes. The class published in the header holds a pointer to each of these classes and delegates responsibility for the corresponding functionality by forwarding the corresponding operations. This is similar to Cheshire Cat, except that some implementation may remain exposed (like `myName` in the example) and there may be more than one helper class. (The helper classes may be defined and implemented in the implementation file – as in the sample code –

Cheshire Cat

```
// cheshire_cat.h  Cheshire Cat –
//           implementation hiding example

#ifndef INCLUDED_CHESHIRE_CAT_H
#define INCLUDED_CHESHIRE_CAT_H
#include <string>
#include <utility>

namespace cheshire_cat {
class telephone_list {
public:
  telephone_list(const std::string& name);
  ~telephone_list();

  std::string get_name() const;

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

  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  class telephone_list_implementation;
  telephone_list_implementation* rep;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
                  const telephone_list& rhs);
};
} // namespace cheshire_cat
#endif
```

or placed in a header file and made available for use by other code.)

Delegation is attractive where there is a distinct area of functionality that can be factored out or shared with another class. In maintenance and performance terms it is similar to Cheshire Cat.

Envelope/Letter

As with Cheshire Cat a private "representation" class is written which implements the same functionality and interface as the naïve class but is defined and implemented entirely within the implementation file. The variations from Cheshire Cat are:

- The "representation" class is derived from the public one.
- The member functions of the public class are declared `virtual` (and overridden in the implementation class).
- The class published in the header holds a pointer to what appears to be another instance of the class but, in fact, is an instance of the derived class.

This is described in some detail in Coplien's "Advanced C++ Style and Idioms" [Cope92].

Frankly Envelope/Letter confuses us – we don't see what advantage it gives over Cheshire Cat. (Maybe it is just a misguided attempt to represent the correspondence of interface and implementation functions explicitly?) But please read Coplien and make up your own mind! In performance terms each client call initiates two function calls dispatched via the v-table – so it is the slowest of the idioms. (However it is rare that the overhead of a virtual function call is significant.)

Interface Class

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

This is described in some detail in Mark Radford's "C++ Interface Classes – An Introduction" [Radford04].

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

Delegation

```
// delegation.h – Delegation implementation hiding
//   example.

#ifndef INCLUDED_DELEGATION_H
#define INCLUDED_DELEGATION_H
#include <string>
#include <utility>

namespace delegation {
class telephone_list {
public:
  telephone_list(const std::string& name);
  ~telephone_list();

  std::string get_name() const;

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

  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  std::string name;
  class dictionary;
  dictionary* lookup;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
                 const telephone_list& rhs);
};
} // namespace delegation
#endif
```

Envelope/Letter

```
// envelope_letter.h – Envelope/Letter
//   implementation hiding example.

#ifndef INCLUDED_ENVELOPE_LETTER_H
#define INCLUDED_ENVELOPE_LETTER_H
#include <string>
#include <utility>

namespace envelope_letter {
class telephone_list {
public:
  telephone_list(const std::string& name);
  virtual ~telephone_list();

  virtual std::string get_name() const;

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

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
protected:
  telephone_list();
private:
  telephone_list* rep;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
                 const telephone_list& rhs);
};
} // namespace envelope_letter
#endif
```

necessary. This is shown as a static `create` *function in the corresponding sidebar.*

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

Note: Interfaces may play an additional role in design to that addressed in this article – they may be used to delineate each of several roles supported by a concrete type. This allows for client code that depend only on (the interface to) the relevant role.

Non-Virtual Public Interface

All member data is removed from the naïve class, the public interface becomes non-virtual forwarding functions that delegate to corresponding private pure virtual functions. As with Interface Class the implementation file defines a derived class that implements these member functions. The derived class is not used directly by client code, which sees only a pointer to the public class.

This is described in some detail in Sutter's "Exceptional C++ Style" [Sut04].

We had thought Non-Virtual Public Interface an idea that had been tried and discarded as introducing unjustified complexity.

Interface Class

```
// interface_class.h – Interface Class
//   implementation hiding example.

#ifndef INCLUDED_INTERFACE_CLASS_H
#define INCLUDED_INTERFACE_CLASS_H
#include <string>
#include <utility>

namespace interface_class {
class telephone_list {
public:
  static telephone_list*
                  create(const std::string& name);
  virtual ~telephone_list()   {}

  virtual std::string get_name() const = 0;

  virtual std::pair<bool, std::string>
  get_number(const std::string& person) const = 0;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number) = 0;
protected:
  telephone_list()   {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(
                const telephone_list& rhs);
};
} // namespace interface_class
#endif
```

While the standard library uses this idiom in the iostreams design we've yet to see an implementation of the library that exploits the additional flexibility (in implementing the public functions) it offers over Interface Class. Further, there are some costs to providing this flexibility:

- *A class definition embodies the contract between code that uses and code that implements that class. By splitting the contract into (public) non-virtual usage and (private) virtual implementation parts it introduces a need to understand both and also a need to document and follow the relationship between them.*
- *There is a development and maintenance cost: because the implementation functions are private to the base class they cannot be called directly by a unit test.*
- *There is a potential performance cost: if the extra function call is not optimised away it can use additional stack space and time.*

Non-Virtual Public Interface

```
// non_virtual_public_interface.h – Non-Virtual
//   Public Interface implementation hiding example

#ifndef INCLUDED_NONVIRTUAL_PUBLIC_INTERFACE_H
#define INCLUDED_NONVIRTUAL_PUBLIC_INTERFACE_H
#include <string>
#include <utility>

namespace non_virtual_public_interface {
class telephone_list {
public:
  static telephone_list* create(
                          const std::string& name);
  virtual ~telephone_list() {}

  std::string get_name() const
    { return do_get_name(); }

  std::pair<bool, std::string>
  get_number(const std::string& person) const
    { return do_get_number(person); }

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number)
    { return do_add_entry(name, number); }
protected:
  telephone_list()    {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(
                const telephone_list& rhs);
  virtual std::string do_get_name() const = 0;
  virtual std::pair<bool, std::string>
  do_get_number(const std::string& person) const = 0;
  virtual telephone_list&
  do_add_entry(const std::string& name,
               const std::string& number) = 0;
};
} // namespace non_virtual_public_interface
#endif
```

Evaluating the Solutions

Problem 1: Reducing Implementation Detail Exposed to the User

All the idioms considered address this problem reasonably successfully. The only implementation detail any of these idioms expose is the mechanism by which they support the separation:

- Interface Class declares virtual functions
- Cheshire Cat exposes a pointer to the "real" implementation
- Non-Virtual Public Interface declares forwarding functions and virtual functions
- Envelope/Letter declares virtual functions and a pointer to the "real" implementation

Delegation is in a way the odd one out, because it does not by nature conceal all the implementation detail. This point is illustrated in our example implementation where the `std::string` member `myName` is visible in the definition of `TelephoneList`. Delegation reduces the implementation noise exposed to clients, but – unless all functionality is delegated to one (or more) other classes – it leaves the class still vulnerable to the problems suffered by the naïve implementation.

Problem 2: Reducing Physical Coupling

When the principal concern is reducing compile time dependencies the size (including indirect inclusions) of the header is more significant than that of the implementation file. However, in most cases, there is very little difference between the header files required by the different idioms – in our example they all have the same includes and the file lengths are as follows:

Cheshire Cat Implementation

```
// MCheshireCat.cpp - implementation hiding example.

#include "cheshire_cat.h"
#include <map>

namespace cheshire_cat {
// Declare the implementation class
class telephone_list::telephone_list_implementation {
public:
  telephone_list_implementation(
                          const std::string& name);
  ~telephone_list_implementation();
  std::string get_name() const;
  std::pair<bool, std::string>
  get_number(const std::string& person) const;
  void add_entry(const std::string& name,
                 const std::string& number);
private:
  typedef std::map<std::string, std::string>
                                      dictionary_t;
  std::string  name;
  dictionary_t dictionary;
};

// Implement the stubs for the wrapper class
telephone_list::telephone_list(
                           const std::string& name)
   : rep(new telephone_list_implementation(name)) {}

telephone_list::~telephone_list() { delete rep; }

std::string telephone_list::get_name() const {
  return rep->get_name();
}

std::pair<bool, std::string> telephone_list::
get_number(const std::string& person) const {
  return rep->get_number(person);
}

telephone_list& telephone_list::add_entry(
                          const std::string& name,
                          const std::string& number) {
  rep->add_entry(name, number);
  return *this;
}

// Implement the implementation class
telephone_list::telephone_list_implementation::
          telephone_list_implementation(
                               const std::string& name)
   : name(name) {}

telephone_list::telephone_list_implementation::
          ~telephone_list_implementation() {}

std::string telephone_list::
   telephone_list_implementation::get_name() const {
  return name;
}

std::pair<bool, std::string>
telephone_list::telephone_list_implementation::
       get_number(const std::string& person) const {

  dictionary_t::const_iterator i
       = dictionary.find(person);

  return(i != dictionary.end()) ?
      std::make_pair(true, (*i).second) :
      std::make_pair(true, std::string());
}

void telephone_list::telephone_list_implementation::
add_entry(const std::string& name,
          const std::string& number) {
  dictionary[name] = number;
}
} // namespace cheshire_cat
```

```
$ wc *.h | sort
   62    163   1580  cheshire_cat.h
   62    184   1677  interface_class.h
   65    162   1535  naive.h
   66    163   1554  delegation.h
   66    164   1605  envelope_letter.h
   94    285   2688  non_virtual_public_interface.h
```

The lack of variation is not surprising: all of the examples have eliminated the `<map>` header file and the only substantial difference is that Non-Virtual Public Interface declares twice as many functions (having both public and private versions of each).

Problem 3: Allowing Customised Implementations

It should be noted that only Interface Class and Non-Virtual Public Interface allow user implementation – the other idioms do not publish an implementation interface.

When our principal concern is that of simplifying the task of implementing the class then the size of the implementation file is most significant:

```
$ wc interface_class.cpp /
                  non_virtual_public_interface.cpp
   85   147  2013  interface_class.cpp
   89   151  2186  non_virtual_public_interface.cpp
```

There is no substantial difference in implementation cost between these approaches as they contain almost identical code.

Conclusion

In scenarios where customisation of implementation needs to be supported the choice is between Interface Class and Non-Virtual Public Interface. In this case we would prefer the simplicity of Interface Class (unless we have a need for the public functions to do more work than forwarding – which leads us into the territory of TEMPLATE METHOD [GOF95]).

Sometimes we wish to develop "value based" classes – these can, for example, be used directly with the standard library containers. Only three of the idioms (Cheshire Cat, Envelope/Letter and Delegation) permit this style of class. (Using value-based classes implies that the identity of class instances is transparent – and that may not be appropriate). Of these options, Cheshire Cat is most often the appropriate choice – although Delegation may be appropriate if it allows common functionality to be factored out.

There are many occasions where user customisation of implementation is not required, and the identity of instances of the class is important. In these circumstances it is reasonable to expect

[concluded at foot of next page]

Interface Class Implementation

```
// MAbstractBaseClass.cpp - implementation hiding
// example.
#include "interface_class.h"
#include <map>

// Declare the implementation class
namespace {
class telephone_list_implementation
   : public interface_class::telephone_list {
public:
  telephone_list_implementation(const std::string& name);
  virtual ~telephone_list_implementation();
private:
  virtual std::string get_name() const;
  virtual std::pair<bool, std::string>
  get_number(const std::string& person) const;
  virtual interface_class::telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
  typedef std::map<std::string, std::string>
                                      dictionary_t;
  std::string name;
  dictionary_t dictionary;
};
} // anonymous namespace

// Implement the stubs for the base class
namespace interface_class {
telephone_list* telephone_list::create(
                          const std::string& name) {
  return new telephone_list_implementation(name);
}
} // namespace interface_class

// Implement the implementation class
namespace {
telephone_list_implementation::
telephone_list_implementation(const std::string& name)
   : name(name) {}

telephone_list_implementation::
~telephone_list_implementation() {}

std::string
telephone_list_implementation::get_name() const
  { return name; }

std::pair<bool, std::string>
telephone_list_implementation::
get_number(const std::string& person) const {
  std::pair<bool, std::string> rc(false,
                                  std::string());
  dictionary_t::const_iterator i
      = dictionary.find(person);
  return(i != dictionary.end()) ?
      std::make_pair(true, (*i).second) :
      std::make_pair(true, std::string());
}

interface_class::telephone_list&
telephone_list_implementation::
add_entry(const std::string& name, const
std::string& number) {
  dictionary[name] = number;
  return *this;
}
} // anonymous namespace
```

Overload Resolution – Selecting the Function

by Mikael Kilpeläinen

Overloading is a form of polymorphism, however, the rules are quite complex in C++. This article tries to explain most of the rules and clarify concepts like the implicit conversion sequence. The main aim is to explain how a function is selected from the set of possibilities. This makes it easier to understand and correct ambiguities the compilers might complain about.

Overview of Overloading Process

Declaring two or more items with the same name in a scope is called *overloading*. In C++ the items which can be overloaded are free functions, member functions and constructors, which are collectively referred to as functions. The compiler selects which function to use at compile time according to the argument list, including the object itself in the case of member functions. The functions that have the same name and are visible in a specific context are called candidates. First the usable functions are selected from the set of *candidates*. These usable functions are called *viable functions*. A function is viable if it can be called, that is the parameter count matches the arguments and an *implicit conversion sequence* exists for every argument to the corresponding parameter. A function having more parameters than there are arguments in an argument list can also be viable if default arguments exist for all the extra parameters. In such cases the extra parameters are not considered for the purpose of overload resolution. Access control is applied after overload resolution, meaning that if the function selected is not accessible in the specified context, the program is ill-formed.

Phases of the function call process:

1. Name lookup
2. Overload resolution
3. Access control

Many different contexts of overloading exist and each has its own set of rules for finding the set of candidate functions and arguments. Those rules are not covered here except for a few important cases which involve a user-defined conversion. After defining the candidates and the arguments for each context, the rest of the overload process is identical for all contexts.

Ordering of Viable Functions

A viable function is better than another viable function if (and only if) it does not have a worse implicit conversion sequence for any of its arguments than the other function and has one of the following properties:

- It has at least one better conversion sequence than the other function.
- It is a non-template and the other function is a template specialisation.
- Both are templates and it is more specialised than the other function according to the partial ordering rules.

The ordering of implicit conversion sequences is explained later. If only one function is better than other functions in the set of viable functions then it is called the *best viable function* and is selected by the overload resolution. Otherwise the call is ill-formed and diagnostics are reported.

Member Functions and Built-in Operators With Overloading

For overload resolution, member functions are considered as free functions with an extra parameter taking the object itself. This is called the *implicit object parameter*. The cv-qualification[1] of the implicit parameter is the same as the cv-qualification of the specified member function. The object is matched to the implicit object parameter to make the overload resolution possible. This is an easy way to make the overloading rules uniform for the member functions and free functions. The implicit object argument is just like other arguments, except for a few special rules: the related conversions cannot introduce temporaries, no user-defined conversions are allowed and an rvalue can be bound to a non-constant reference. For static member functions the implicit object parameter is not considered since there is no object to match it. Also the built-in operators are considered free functions for the purpose of overload resolution.

Examples:

```
struct type {
  void func(int) const;
  void other();
};
```

1 `const` and `volatile` are the cv-qualifiers.

[continued from previous page]

client code to manage object lifetime explicitly (e.g. by using a smart pointer). Both Interface Class and Cheshire Cat are reasonable choices here. Interface Class is simpler, but where a strong separation of interface and implementation is required Cheshire Cat may be preferred.

Alan Griffiths & Mark Radford

References

[WEB05] `http://www.octopull.demon.co.uk/c++/implementation_hiding.html`

[Cope92] J. Coplien. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[Sut00] Herb Sutter. *Exceptional C++*, Addison-Wesley, 2000

[Griffiths99] `http://www.octopull.demon.co.uk/c++/TheGrin.html`

[Radford04] Mark Radford, "C++ Interface Classes – An Introduction", *Overload 62*, and also available from `http://www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf`

[Sut04] Herb Sutter. *Exceptional C++ Style*, Addison-Wesley, 2004

[GOF95] Gamma, Helm, Johnson & Vlissides. *Design Patterns*, Addison-Wesley, 1995

Acknowledgments

Thanks to Tim Penhey and Phil Bass for commenting on drafts of this article.

The member functions are considered as

```
void func(type const&, int);
void other(type&);

char* p;
p[0];
```

The subscript operator is considered as

```
T& operator[](T*, ptrdiff_t);
```

where T is a cv-(un)qualified type

Conversions

The implicit conversion sequences are based on single conversions. These simple implicit conversions provide a great deal of flexibility and can be helpful if used correctly. Even though single conversions are quite easy, the interaction between the sequences of conversions and the overloading is far from simple. The standard conversions are the built-in conversions, those are categorised and ranked to form an intuitive order. This is the basis for ranking the conversion sequences consisting of only standard conversions. There are three ranks for these conversions (see Table 1). In addition to those standard conversions, a derived-to-base conversion exists but only in the description of implicit conversion sequences. It has a conversion rank.

Conversion	Category	Ranking	Rank
No conversions required	Identity	Exact match	1
Lvalue-to-rvalue conversion Array-to-pointer conversion Function-to-pointer conversion	Lvalue transformation		
Qualification conversion	Qualification adjustment		
Integral promotions Floating point promotions	Promotion	Promotion	2
Integral conversions Floating point conversions Floating-integral conversions Pointer conversions Pointer to member conversions Boolean conversions	Conversion	Conversion	3

Table 1: Standard Conversions (smallest number is highest rank)

Examples:
`char` → `int` (integral promotion)
`float` → `long` (floating-integral conversion)
type → type `const` (qualification conversion)
type → type (identity conversion)

Besides standard conversions there are the user-defined conversions, meaning conversion functions and converting constructors. User-defined conversions are applied only if they are unambiguous. It is good to know that at most one user-defined conversion is implicitly applied to a single value. Three forms of conversion sequences can be constructed from these different conversions:

- Standard conversion sequence
- User-defined conversion sequence
- Ellipsis conversion sequence

Standard Conversion Sequences

The standard conversion sequence is either an identity conversion or consists of one to three standard conversions from the four categories when identity is not considered, at most one conversion per category. The standard conversions are always applied in a certain order: Lvalue transformation, Promotion or Conversion and Qualification adjustment. The standard conversion sequence is ranked according to the conversions it contains, the conversion with the lowest rank dictates the rank of the whole sequence.

Examples:
`bool` → `short` (conversion rank)
`char` → `char const` (exact match rank)
`char` → `int` → `int const` (promotion rank)
`float[]` → `float*` → `float const*` (exact match rank)

User-Defined Conversion Sequences

A user-defined conversion sequence is a composition of three pieces: first an initial standard conversion sequence followed by a user-defined conversion and then followed by another standard conversion sequence. In the case when the user-defined conversion is a conversion function, the first conversion sequence converts the source type to the implicit object parameter so that the user-defined conversion can be applied.

On the other hand, if the user-defined conversion is a constructor, the source type is converted to a type required by the constructor. After the user-defined conversion is applied, the second standard conversion sequence converts the result to a destination type. If the user-defined conversion is a template conversion function, the second standard conversion sequence is required to have an exact match rank. A conversion from a type to the same type is given an exact match rank even though a user-defined conversion is used. This is natural when passing parameters by value and hence using a copy constructor.

Examples:

```
   struct A { operator int(); };
   long var = A();
```

`A` → `int` → `long`

```
   struct B { B(float); };
   void func(B const&);
   func(0);
```

`int` → `float` → `B` → `B const`

Ellipsis Conversion Sequences

The last and third form of conversion sequence is an ellipsis conversion sequence, which happens when matching an argument to an ellipsis parameter.

Examples:

```
void func(...);
func(0); // an ellipsis conversion sequence,
         // int matching to an ellipsis
         // parameter.
```

Reference and Non-Reference Parameters

If a parameter type is not a reference, the implicit conversion sequence models a copy-initialisation. In that case any difference in top level cv-qualification is not considered as a conversion. Also the use of a copy constructor is not ranked as a user-defined conversion but as an exact match and hence is not a conversion. However, if the parameter is a reference, binding to a reference occurs. The binding is considered an identity conversion and hence if the destination type binds directly to the source expression, it is an exact match. An rvalue can not be bound to a non-`const` reference and a candidate requiring such is not viable. If the type of the argument does not directly bind to the parameter, the implicit conversion sequence models a copy-initialisation of a temporary to the underlying type of the reference, similar to the case of a non-reference.

Basic Ordering of Conversion Sequences

The implicit conversion sequences for the n^{th} parameters of the viable functions need to be ordered to select the best viable function if one exists. The three basic forms of sequences are ordered so that the standard conversion sequence is better than the user-defined conversion sequence and the user-defined conversion sequence is better than the ellipsis conversion sequence. In case that two conversion sequences cannot be ordered, they are said to be indistinguishable. This is rather easy and intuitive ordering but there is a lot more to it.

Ordering of Standard Conversion Sequences

Standard conversion sequences are ordered by their rank. The higher the rank, the better the sequence. Another important ordering is that a proper subsequence of another sequence is better than the other sequence. The comparison excludes lvalue transformations. An identity conversion is considered to be a subsequence of any non-identity conversion sequence. Also there are other rules that apply with the standard conversion sequences: If two sequences have the same conversion rank, they are indistinguishable unless one is a conversion of a pointer to `bool` which is a worse conversion than other conversions. In case of converting a type to its direct or indirect base class, the conversion to a base class closer in the inheritance hierarchy is a better conversion than a conversion to a base class that is further away. The same applies with pointers and references, also with pointers `void*` is considered to be the furthest in the hierarchy.

Ordering of User-Defined Conversion Sequences

User-defined conversion sequences are somewhat more difficult to order. Constructing a user-defined conversion sequence for a specific parameter means first using the overload resolution to select the best user-defined conversion for the sequence. This works just like ordinary overloading but now the first parameter of a converting constructor is considered as a destination type and similarly in the case of a conversion function the implicit object parameter. In case there is more than one best user-defined conversion, the second standard conversion sequence is used to decide which conversion sequence is better than the other. If there is no best conversion sequence for that specific parameter, the sequence is an *ambiguous conversion sequence*. It is treated as any user-defined conversion sequence because it always involves a user-defined conversion. The purpose of an ambiguous conversion sequence is to keep a specific function viable. Removing the function from the set of viable functions could cause some other function to become the best viable function even if it clearly is not. If a function using an ambiguous conversion sequence is selected as the best viable function, the call is ill-formed.

Examples:

```
struct A;
struct B {
  B(A const&);
};

struct A {
  operator B() const;
  operator int() const;
};
void func(B);
void func(int);

func(A());
```

The call is ambiguous, however, the parameter `B` has an ambiguous conversion sequence and if the function having this parameter was eliminated the call would not be ambiguous. This is because there would be only one function to select.

For each argument the implicit conversion sequences are constructed. After that the sequences are compared and ordered. Two user-defined conversion sequences are indistinguishable unless they use the same user-defined conversion in which case the second standard conversion sequence is conclusive.

Difficulties With User-Defined Conversions

There are a few oddities with user-defined conversions, mostly when the destination type is a reference.

One such context is an initialisation by conversion function for direct reference binding. This means that a conversion function converting to a type which is reference-compatible with the destination type exists. In this case the candidates for selecting the user-defined conversion are only the conversion functions returning a reference that is compatible with the destination reference.

Another thing is that in the same context, the second standard conversion sequence is considered to be an identity conversion if the result binds directly to the destination, or a derived-to-base conversion in the case of a base class. This means for example

[concluded at foot of next page]

Digging a Ditch
Writing a Custom Stream

by Paul Grenyer

Writing a custom stream is easy! Most people are now entirely comfortable using `std::vector` and `std::list`, and know the difference between a `std::map` and a `std::set`. However, the use and extension of the C++ standard library's streams is still considered difficult.

In this article I am going to look at writing a logging stream. A logging stream inserts the current date and time at the beginning of a buffer full of characters when it is flushed. The buffer is flushed to another stream which can modify the characters further or write them, for example, to the console (`std::cout`) or to a file (`std::ofstream`).

In section 13.13.3 of The C++ Standard Library [Josuttis] Nico Josuttis discusses how to write a custom stream in a fair amount of detail. Even though the book is widespread among developers, the section on streams does not appear to be widely read. Therefore in this article I am going to follow reasonably closely the line that Josuttis takes, but will cut out a lot of the unnecessary background which may scare the people who, wrongly, feel it must be read and understood before embarking on a custom stream. I will also discuss and resolve a potential initialisation problem not explored by Josuttis.

Stream Buffer

The heart of a stream is its buffer. Buffer is a misnomer as it does not have to buffer at all and can, if it so chooses, process the characters immediately.

Along with buffering, if required, the stream buffer does all the reading and writing of characters for the stream. The standard library provides `std::basic_streambuf` as a base class for stream buffers. Listing 1 shows a stream buffer that converts all the characters streamed to it to upper case and writes them with `putchar`:

```
#include <streambuf>
#include <locale>
#include <cstdio>

template<class charT,
         class traits = std::char_traits<charT> >
class outbuf
     : public std::basic_streambuf<charT, traits> {
private:
  typedef typename std::basic_streambuf<charT,
                       traits>::int_type int_type;

  // Central output function.
  // - print characters in uppercase.
  virtual int_type overflow(int_type c) {

    // Check character is not EOF
    if(!traits::eq_int_type(c, traits::eof())) {

      // Convert character to uppercase.
      c = std::toupper<charT>(c,
                       std::basic_streambuf<charT,
                              traits>::getloc());
```

[continued from previous page]

that there is no ordering for different cv-qualifications. The rules concerning this might change in future standards to make the rules consistent and to meet one's expectations.

Another Way to Handle User-Defined Conversion Sequences

Considering the overload rules for user-defined conversions, it is easy to notice that the selection of the user-defined conversion can be combined with the rest of the overload process. This leads to a few rules:

If the destination parameter is the same for two sequences, the first standard conversion sequences are used to order these user-defined conversion sequences.

After that the second standard conversion sequence is used to select the best conversion sequence.

Of course one has to be careful not to mix those with the conversion sequences that do not have the same destination.

Function Templates With Overloading

In most cases a function template behaves just like a normal function when considering overload resolution. The template argument deduction is applied, if it succeeds, the function is added to the candidates set. Such a function is handled like any other function, except when two viable functions are equally good, the non-template one is selected. In case both are a specialisation of a function template, partial ordering rules are applied. The partial ordering rules are out of the scope of this article.

Conclusion

This just about covers all there is to know about conversion sequences. However there are a lot of subjects to cover which are related to the subject of this article, to mention a few: finding candidate sets, overloadable declaration and partial ordering. It can be somewhat hard to remember all the rules related to the issue, however, only a subset is normally needed. The basic ideas are easy enough to remember and those are the ones usually needed and of course it is always possible to look up the exact rules.

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Acknowledgements

Thank you to Rani Sharoni, Terje Slettebø, Stefan de Bruijn and Paul Grenyer for providing important comments.

References

[1] ISO/IEC 14882-2003, *Standard for the C++ language*

[2] David Vandevoorde and Nicolai M. Josuttis, *C++ templates: The Complete Guide*, Addison Wesley 2002

```
      // Write character to standard output
      if(putchar(c) == EOF) {
        return traits::eof();
      }
    }

    return traits::not_eof(c);
  }
};
```

Listing 1: Example stream buffer

The overflow member function of `std::basic_streambuf` is called for each character that is sent to the stream buffer. Overriding it allows the behaviour to be modified. The example in Listing 1 above performs the following for each character sent to overflow:

1. The character is tested to make sure it is not an indication of the end of a file or an error.
2. The character is converted to uppercase.
3. The character is written to standard out. If an error occurs while writing the character this is indicated by returning `traits::eof()`.
4. An indication of whether or not the character represents the end of a file or an error is returned.

Traits are used throughout Listing 1 to ensure that EOF is detected and handled correctly. Streams can be used with any character type that has a corresponding set of character traits. A detailed knowledge of character traits is not required when using the built in character types `char` and `wchar_t` as their traits are already part of the standard library. Character traits are discussed in 14.1.2 of Josuttis.

Output Stream

The easiest way to use a stream buffer is to pass it to an output stream as shown in Listing 2 below:

```
#include <streambuf>
#include <ostream>
#include <locale>
#include <cstdio>

template<class charT,
         class traits = std::char_traits<charT> >
class outbuf : public std::basic_streambuf<charT,
                                           traits> {
private:
  typedef typename std::basic_streambuf<charT,
                        traits>::int_type int_type;

  // Central output function.
  // - print characters in uppercase.
  virtual int_type overflow(int_type c) {
    // Check character is not EOF
    if(!traits::eq_int_type(c, traits::eof())) {
      // Convert character to uppercase.
      c = std::toupper<charT>(c,
                        std::basic_streambuf<charT,
                                traits>::getloc());
      // Write character to standard output
      if(putchar(c) == EOF) {
        return traits::eof();
      }
    }

    return traits::not_eof(c);
  }
};

int main() {
  outbuf<char> ob;
  std::basic_ostream<char> out(&ob);
  out << "31 hexadecimal: "
      << std::hex
      << 31 << std::endl;
  return 0;
}
```

Listing 2: Passing a stream buffer to an output stream

The output from the example in Listing 2 is:

```
31 HEXADECIMAL: 1F
```

The example in Listing 2 demonstrates a working stream, but is not an ideal solution as the stream buffer must be declared separately from the stream itself. A common solution is to create a subclass of `std::basic_ostream` with the stream buffer as a member which can be passed to the `std::basic_ostream` constructor as shown in Listing 3:

```
template<class charT,
         class traits = std::char_traits<charT> >
class ostream
       : public std::basic_ostream<charT, traits> {
private:
  outbuf<charT, traits> buf_;

public:
  ostream() : std::basic_ostream<charT,
                          traits>(&buf_), buf_() {}
};
```

Listing 3: Subclass of `std::basic_ostream`

Having the stream buffer as a member introduces a potential initialisation problem. The solution to the problem introduces a further problem hidden deep within the C++ standard [C++ Standard]. However, this second problem is also easily fixed.

Problem 1

If the stream buffer is dereferenced in `std::basic_ostream`'s constructor or in its destructor, undefined behaviour can occur as the stream buffer will not have been initialised. At least one well known and widely used standard library implementation does nothing to avoid this and does not need to. Library implementers know their stream implementations and whether or not protection is needed. We, as stream extenders writing for potentially any number of different stream implementations, do not. There is no guarantee in the C++ standard to fall back on either.

Josuttis places the buffer before `std::basic_ostream`'s constructor in the initialisation list, which makes no difference at all as stated in 12.6.2/5 of the C++ standard:

Initialization shall proceed in the following order:

- *First, and only for the constructor of the most derived class as described below, virtual base classes shall be initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where "left-to-right" is the order of appearance of the base class names in the derived class base-specifier-list.*
- *Then, direct base classes shall be initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers).*
- *Then, nonstatic data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).*
- *Finally, the body of the constructor is executed.*

Note: the declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization.

The fact that the stream buffer is not initialised before it is passed to `std::basic_ostream`'s constructor may not cause a problem with your compiler and library, but why risk it when there is a simple and straightforward solution? On the other hand, it may fail in a screaming fit immediately. Moving the stream buffer to a private base class which is initialised *before* `std::basic_ostream` solves the problem nicely. The initialisation order of base classes *is* specified as stated in 12.6.2/5 above. Listing 4 shows the base class which is used to initialise the stream buffer and how to use it with the output stream.

```
template<class charT,
         class traits = std::char_traits<charT> >
struct outbuf_init {

private:
  outbuf<charT, traits> buf_;

public:
  outbuf<charT, traits>* buf() {
    return &buf_;
  }
};

template<class charT,
         class traits = std::char_traits<charT> >
class ostream : private outbuf_init<charT, traits>,
         public std::basic_ostream<charT, traits> {

private:
  typedef outbuf_init<charT, traits> outbuf_init;

public:
    ostream() : outbuf_init(),
       std::basic_ostream<charT,
                    traits>(outbuf_init::buf()) {}
};
```

Listing 4: Initialising the stream buffer

Problem 2

`basic_ios` is a virtual base class of `basic_ostream`. The C++ standard (27.4.4/2) describes its constructor as follows:

> ***Effects:*** *Constructs an object of class* `basic_ios` *(27.4.2.7) leaving its member objects uninitialized. The object must be initialized by calling its* `init` *member function. If it is destroyed before it has been initialized the behavior is undefined.*

`basic_ios::init` is called from within `basic_ostream`'s constructor. This is where things get complicated. As `basic_ios` is a virtual base class of `basic_ostream`, the objects which make up an `ostream` object are initialised in the following order (see 12.6.2/5):

```
...
basic_ios
outbuf
outbuf_init
basic_ostream
ostream
```

Therefore the constructors of `basic_ios` and `outbuf` are both called before the constructor of `basic_ostream` and therefore before `basic_ios::init` is called. This means that if the `outbuf` constructor throws an exception, `basic_ios`'s destructor will be called before `basic_ios::init`; resulting in the undefined behaviour described in 27.4.4/2.

The answer to this problem is contained within 12.6.2/5 and is very simple. Making `ostream` inherit *virtually*, as well as privately, from `outbuf_init` causes it to be constructed before anything else:

```
template<class charT,
         class traits = std::char_traits<charT> >
class ostream
      : private virtual outbuf_init<charT, traits>,
        public std::basic_ostream<charT, traits> {

private:
  typedef outbuf_init<charT, traits> outbuf_init;

public:
  ostream()
      : outbuf_init(),
        std::basic_ostream<charT,
                    traits>(outbuf_init::buf()) {}
};
```

The initialisation order then becomes:

```
outbuf
outbuf_init
...
basic_ios
basic_ostream
ostream
```

Now, if `output_buf` does throw an exception there is no undefined behaviour as the `basic_ios` has not yet been created.

ostream can be made easier to use by introducing a couple of simple typedefs for common character types:

```
typedef ostream<char> costream;
typedef ostream<wchar_t> wostream;

int main() {
  costream out;
  out << "31 HEXADECIMAL: " << std::hex
      << 31 << std::endl;
  return 0;
}
```

Listing 5: Typedefs for using ostream

That completes the implementation for the simplest possible custom stream.

Logging Stream Buffer

The previous example of a stream buffer was very basic, potentially inefficient and didn't actually buffer the characters streamed to it. The logging stream mentioned at the start of this article requires the characters to be buffered. When the buffer is flushed the time and date are prepended before it is passed on to the next stream.

Josuttis also has an example of a buffered stream buffer. However, his example uses a fixed array for a buffer that gets flushed when it is full. The logging stream should only flush the buffer when instructed to do so, with a std::endl or a call to flush. To accomplish this, the fixed array can be replaced with a std::vector.

As already mentioned the logging stream simply buffers the characters streamed to it and passes them on to another stream, preceded by a time and date, when flushed. Therefore the stream buffer must contain some form of reference to the other stream.

Listing 6 shows a basic implementation for the logging stream buffer. A std::vector based buffer has been introduced and overflow modified to check for EOF before inserting its character into the buffer.

```
#include <streambuf>
#include <vector>

template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf
     : public std::basic_streambuf<charT, traits> {
private:
  typedef typename std::basic_streambuf<charT,
                       traits>::int_type int_type;
  typedef std::vector<charT> buffer_type;
  buffer_type buffer_;

  virtual int_type overflow(int_type c) {
    if(!traits::eq_int_type(c, traits::eof())) {
      buffer_.push_back(c);
    }
    return traits::not_eof(c);
  }
};
```

Listing 6: Basic implementation of logging stream buffer

As it stands the stream buffer in Listing 6 only buffers characters. It never flushes them. A pointer to an output stream buffer, that the characters can be flushed to, is required. The initialisation and undefined behaviour fixes described in the previous section have the side effect that logoutbuf will be a member of a virtual base class and therefore should have a default constructor. A virtual base class constructor must be called explicitly or implicitly from the constructor of the most derived class (12.6.2/6). A default constructor eliminates the need for explicit constructor calling. This in turn means that a reference to an output stream cannot be passed in through the constructor and therefore a pointer to the output stream buffer must be stored instead and initialised by way of an initialisation function. This is not ideal, but a trade-off to guarantee safety elsewhere. The initialisation function is also in keeping with the buffer initialisation in basic_ios.

```
template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf
     : public std::basic_streambuf<charT, traits> {
private:
  typedef typename std::basic_streambuf<charT,
                       traits>::int_type int_type;

  typedef std::vector<charT> buffer_type;

  std::basic_streambuf<charT, traits>* out_;
  buffer_type buffer_;

public:
  logoutbuf() : out_(0), buffer_() {}
  void init(std::basic_ostream<charT,
                               traits>* out) {
    out_ = out;
  }
  ...
};
```

Listing 7: Initialising the output stream buffer

Listing 7 shows the logoutbuf stream buffer with the output stream buffer pointer and initialisation function. A constructor has also been added to make sure that the output stream buffer pointer is initialised to 0, so that it can be reliably checked before characters are sent to it.

When basic_ostream::flush is called, either directly or via std::endl, it starts a chain of function calls that finally results in basic_streambuf::sync being called. This is where the buffer should be flushed. The buffer should also be flushed when a logoutbuf object is destroyed, so sync should also be called from the logoutbuf destructor.

```
template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf
     : public std::basic_streambuf<charT, traits> {
  ...
public:
  ...
```

```
  ~logoutbuf() {
    sync();
  }
  ...

private:
  ...
  virtual int sync() {
    if(!buffer_.empty() && out_) {
      out_->sputn(&buffer_[0],
                  static_cast<std::streamsize>
                                (buffer_.size()));
      buffer_.clear();
    }
    return 0;
  }
};
```

Listing 8: Synchronising the buffer

Listing 8 shows the implementation of the `sync` function. It checks the buffer to make sure there is something in it to flush and then checks the output stream buffer pointer to make sure the pointer is valid. The contents of the buffer are then sent to the output stream buffer, via its `sputn` function, and then cleared.

`basic_streambuf`'s `sputn` function takes an array of characters as its first parameter and the number of characters in the array as its second parameter. `std::vector` stores its elements contiguously in memory, like an array, so the address of the first element in the buffer can be passed as `sputn`'s first parameter. `std::vector`'s size function is used to determine the number of elements in the buffer and can therefore be used as `sputn`'s second parameter. The type of `sputn`'s second argument is the implementation defined typedef `std::streamsize`. As the return type of `std::vector::size` is also implementation defined (and not necessarily the same type), `sputn`'s second parameter must be cast to avoid warnings from compilers such as Microsoft Visual C++. There is a possibility that the number of characters stored in the buffer will be greater than `std::streamsize` can hold, but this is highly unlikely.

`logoutbuf` is now a fully functioning, buffered output stream buffer and can be plugged into a `basic_ostream` object and tested.

```
...
int main() {
  logoutbuf<char> ob;
  ob.init(std::cout.rdbuf());

  // Flush to std::cout
  std::basic_ostream<char> out(&ob);
  out << "31 hexadecimal: " << std::hex
      << 31 << std::endl;
  return 0;
}
```

Listing 9: Using `logoutbuf`

Listing 9 creates a `logoutbuf` object, sets `std::cout`'s stream buffer as its output stream buffer and then passes it to a `basic_ostream` object, which then has character streamed to it. The output from the example in Listing 9 is:

```
31 hexadecimal: 1f
```

The next step is to generate the time and date that will be flushed to the output stream buffer prior to the contents of the `logoutbuf` buffer. The different ways of generating a date and time string are beyond the scope of this article so I am providing the following implementation, which will handle both `char` and `wchar_t` character types, without any explanation beyond the comments in the code:

```
#include <streambuf>
#include <vector>
#include <ctime>
#include <string>
#include <sstream>

...

template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf
     : public std::basic_streambuf<charT, traits> {
  ...
private:
  std::basic_string<charT, traits> format_time() {
    // Get current time and date
    time_t ltime;
    time(&ltime);

    // Convert time and date to string
    std::basic_stringstream<charT, traits> time;
    time << asctime(gmtime(&ltime));

    // Remove LF from time date string and
    // add separator
    std::basic_stringstream<char_type> result;
    result << time.str().erase(
                time.str().length() - 1) << " - ";

    return result.str();
  }
  ...

  virtual int sync() {
    if(!buffer_.empty() && out_) {
      const std::basic_string<charT, traits> time
                                  = format_time();
      out_->sputn(time.c_str(),
                  static_cast<std::streamsize>
                                  (time.length()));
      out_->sputn(&buffer_[0],
                   static_cast<std::streamsize>
                                (buffer_.size()));
      buffer_.clear();
    }
    return 0;
  }
  ...
};
```

Listing 10: Adding date and time

The sync function in Listing 10 now sends a date and time string (plus the separator) to the output stream buffer before flushing the logoutbuf buffer. The result of running the example from Listing 9 is now:

```
Fri Apr 20 16:00:00 2005 - 31 hexadecimal: 1f
```

logoutbuf is now fully functional, but there is a further modification that can be made for the sake of efficiency. Currently overflow is called for *every single character* streamed to the stream buffer. This means that to stream the 31 hexadecimal: string literal to the stream buffer involves *16* separate function calls. This can be reduced to a single function call by overriding xsputn.

```
...
#include <algorithm>

template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf
     : public std::basic_streambuf<charT, traits> {
  ...
private:
  ...
  virtual std::streamsize xsputn(const char_type* s,
                                 std::streamsize num) {
    std::copy(s, s + num,
         std::back_inserter<buffer_type>(buffer_));
    return num;
  }
  ...
};
```

Listing 11: Overriding xsputn

xsputn takes the same parameters as basic_streambuf::sputn and uses the std::copy algorithm together with std::back_inserter to insert the characters from the array into the buffer. logoutbuf is now complete.

logoutbuf does of course require its own logoutbuf_init class and basic_ostream subclass, with a few modifications:

```
template<class charT,
         class traits = std::char_traits<charT> >
class logoutbuf_init {
private:
  logoutbuf<charT, traits> buf_;

public:
  logoutbuf<charT, traits>* buf() {
    return &buf_;
  }
};

template<class charT,
         class traits = std::char_traits<charT> >
class logostream
       : private virtual logoutbuf_init<charT,
                                        traits>,
         public std::basic_ostream<charT, traits> {
private:
  typedef logoutbuf_init<charT, traits>
                                     logoutbuf_init;
public:
  logostream(std::basic_ostream<charT,
                                traits>& out)
      : logoutbuf_init(),
        std::basic_ostream<charT,
                   traits>(logoutbuf_init::buf()) {
    logoutbuf_init::buf()->init(out.rdbuf());
  }
};

typedef logostream<char> clogostream;
typedef logostream<wchar_t> wlogostream;
```

Listing 12: logoutbuf_init class and basic_ostream subclass

The logoutbuf_init class is actually the same as the one form the previous section; it's the logostream that is slightly different. The constructor takes a single parameter which is the output stream and its body passes its stream buffer to logoutbuf via init (suddenly the trade off doesn't seem so bad).

The final test example is shown in Listing 13:

```
...
int main() {
  costream out(std::cout);
  out << "31 hexadecimal: " << std::hex
      << 31 << std::endl;
  return 0;
}
```

Listing 13: Using the stream

Conclusion

The stream buffer is clearly the heart of an output stream. The potential for a stream buffer being accessed before it is initialised is easily avoided, as is the possibility of undefined behaviour, with the minimal of tradeoffs.

The buffering of characters streamed to a stream buffer is easily handled by a std::vector with no need for extra memory handling. Multiple characters can be added to a std::vector just as easily as single characters and the contiguous memory elements make it easy to flush to an output stream.

Writing a custom stream is *easy*! I believe this article shows just how easy it is, even with a minimum of background knowledge.

Paul Grenyer
paul@paulgrenyer.co.uk

References

[Josuttis] Nicolai M. Josuttis, *The C++ Standard Library*, Addison-Wesley, ISBN: 0-201-37926-0.

[C++ Standard] *The C++ Standard*, John Wiley and Sons Ltd, ISBN: 0-470-84674-7

Acknowledgments

Alisdair Meredith, Alan Stokes, Jez Higgins, Alan Griffiths, Thaddaeus Frogley.