contents

credits & contacts

Overload Editor:
John Merrells
overload@accu.org

Contributing Editor:
Alan Griffiths
alan@octopull.demon.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

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

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial - What I Want For Christmas

As some of you know, I've had an eventful year, and this seems like a suitable opportunity to reflect upon those aspects of it that relate to software development. Things are always changing and there is often a pattern but one sometimes needs to step back a bit to see it. This time I'm going to take a step back thirty odd years and describe the way that software development happened then.

When I first started writing programs I'd write them out on paper, then when I'd checked it over I'd transcribe each line by punching holes into a piece of card (very carefully – most errors required starting the card again). When I had completed all the cards I'd use several elastic bands to ensure that the stack of cards was secured in order and place it into a tray with other programs in the same format. Later that day the tray would be carried to a local computer centre and the programs transferred to other trays, a card reader and eventually united with corresponding printouts and placed into a tray awaiting transfer back from the computer centre. If I timed things right a program could be turned around updated and resubmitted a second time in the same day!

The effect of this was that a lot of the coding of a program was concentrated into a few intensive ten-minute intervals separated by hours of suspense. All too often this meant that a mistake was made in the rush, but not noticed until the stack of cards had begun its tortuous journey to the computer centre and back again. It may sound horribly inefficient to the current generation but programs were really developed this way. Nowadays errors that would not have been reported by the compiler for half a day or more are highlighted on the screen before I even save the file!

Although people spent a lot of effort trying to make the "dead time" more effective, any attempt to improve efficiency by working on several programs at once never really worked as well as might be hoped. The relentless cycle of turnarounds forced the development cycles into synchronisation, and as there was always one program that was more urgent than the rest it stole the time that the others needed. And, while I've focussed on writing code, it wasn't just getting the program to compile that was like this, testing and deployment followed similar processes.

But code got written and systems got delivered.

The underlying difference between the way things were then and the way things are now is the speed of feedback. Most readers will be familiar with development environments that highlight syntax errors as you type – problems that could once have led to days of delays and frustration are detected and corrected without conscious thought. Such a change doesn't only affect the speed of progress, it also changes the way that we approach the task. Even those readers without this facility will be working in an environment where it is more effective to use a compiler to check syntax than it is to do so "by hand".

Having reliable and immediate feedback available provides a level of confidence that allows the developer's attention to focus elsewhere. (This is just as well, because the effect of having better tools isn't that the job has got easier – the range of problems that we are willing to tackle has expanded to compensate.)

Naturally, there is much more to developing software than getting the syntax of the code right, and much of this is also dependent upon accuracy. And there are two approaches to accuracy: avoidance of error and correction of error. Each can be appropriate in the right circumstances and, as I have tried to illustrate in the context of coding, the choice can depend upon the tools available.

Traditionally, software development processes have been based around avoidance of errors: getting the requirements right and big up front design all comes from an era of slow, inefficient feedback. There is a significant cost to manually double and triple checking everything to reduce the errors being fed into a process. Automated error detection that provides early feedback and allows early correction is often much more effective. And, based on my experiences this year, I think that it is becoming available for many more aspects of software development.

Unit Testing

The checking of individual units of development is the province of unit tests, and the ability to run these automatically as part of the development environment is just about there for some development technologies. For example, there are free JUnit "plug-ins" for most of the popular Java development environments. Having tests light up "green" (for success) or "red" (for failure) when changes are made can trap a lot of silly errors soon enough after they are made that they don't disrupt a developer's line of thought any more than the occasional compiler error. Of course, as yet, this isn't quite as widespread as syntax checking editors – for example, I don't know of a CppUnit plug-in for the Visual Studio environment my current client favours. So, number one on my "Christmas list" is the availability of such a tool for any environment that I happen to be working in.

Naturally, there are additional issues with the use of unit tests, such as persuading both developers and management of their usefulness. This can be a significant problem: there is a cost to both writing unit tests and to running them – and they do not detect all errors. Much as I would like to I cannot point to scientific comparisons between "equivalent" projects run with and without unit tests that demonstrate the benefits. All I can give is anecdotal evidence that the projects on which I've been able to instil a culture of unit testing have had far fewer problems when it came to integration and delivery. (But unit tests are far from the only change that I've introduced – and projects can be delivered successfully without them.)

The one thing that I can say about having unit tests in place is that the level of rework is much lower. As one developer put it: "it is a pain writing these unit tests – but I like getting things right first time”. But it isn’t as simple as that: things are not always “right first time” – sometimes the requirements have been misunderstood (or have changed: not only can the business change, but the process of capturing requirements can question assumptions, and delivering a software system can offer unexpected alternative approaches).

While there have been attempts to catalogue and collate development practices that work there is very little convincing evidence for many of the things that I would like to believe. Of course, when working with like-minded individuals this isn’t an issue (credible claims require little evidence), but when trying to justify and motivate change, it can be a major problem. When talking to management and developers who believe that standardisation of process, or a new technology, or some other “magic bullet” is the answer to all their development woes then any claims I make will not be considered credible without substantial evidence. So that is the next item on my list: citable evidence of the effectiveness and applicability of alternative development practices.

Functional Testing

Some time ago I came across one of Ward Cunningham’s innovations: “Fit”. Fit is a Java framework for describing system functionality as a web page that can be executed against the system under development. It requires the developers to write some lightweight “fixture” classes that map the requirements embedded in the web page to interactions with the system. The fact that the requirements can be executed directly does a lot to address the ambiguities that frequently find their way into the testing of functional requirements.

More recently (at The Extreme Tuesday Club) I came across some work that builds upon the Fit framework. “Fitnesse”, produced by ObjectMentor, is a Wiki implemented around the Fit framework that facilitates the capture of functional requirements in an executable form: as Fit webpages. Michael Feathers (of ObjectMentor) has also produced FitCpp – a C++ implementation of the Fit framework. (*There are some bugs and other issues to resolve with FitCpp but I’ve been working with it (and Fitnesse) for my current client and, assuming I get suitable permissions, I will have put the resulting material on my website by the time you read this editorial.*)

One of the great things about this approach is that there is very easy visibility of project process. One may set up a summary webpage that lists all the functional tests, colour coded according to whether the functionality is available (green), is failing (red) or has yet to be addressed (grey). Because executing the tests directly against the system produces these results the feedback is always immediate, up to date and honest (which avoids the temptation to exaggerate progress – both to oneself and to others).

It is easy to overlook what this means to people outside the development group. All too often their experience of software development resembles the coding process I described above: concentrated effort at the beginning with lots of effort invested in getting it right, followed by things being “out of their hands” for a long period before the results are visible. It is only then that mistakes, ambiguities and misunderstandings become apparent. Publishing the current state of development on the intranet gives them much needed feedback early in the development cycle. And, because it is a Wiki, it is simple for the requirements to be updated. And because the requirements are the functional tests these too are maintained in a single, authoritative, place.

Fitnesse demonstrates that it is possible to bring requirements capture and functional testing much closer together than has ever been my experience in the past. This (or something like it but better) should be part of the toolkit on any project. Another one for my Christmas list!

Refactoring

It has been a few years since Martin Fowler codified a number of coding practices that experienced developers know are needed but are hard to associate with a quantifiable benefit. These “refactorings” are transformations that leave the functionality unchanged but make the structure of the code more amenable to further development. In the Java world there is now widespread support for automating these transformations.

These facilities are great: it doesn’t sound much but, to take one example, being able to remove a block of code from a growing method body by selecting it, choosing “extract method” from the menu and then entering the method name is so much simpler than the “old way”. The developer is freed from the tedium and mistakes of copying the code, changing the indentation, working out what the parameters need to be and what the return type needs to be (and occasionally discovering that there are subtle reasons why the code cannot be moved after all).

I’ve yet to encounter corresponding support for C++ developers – which is understandable (both in its compilation model and its syntax C++ is a much harder language to address than Java). But this is my list and I see no reason to be reasonable in my demands: these facilities are great and I don’t want C++ to be left out.

Alan Griffiths
alan@octopull.demon.co.uk
www.octopull.demon.co.uk

Copy Deadlines

All articles intended for publication in *Overload 59* should be submitted to the editor by January 1st 2004, and for *Overload 60* by March 1st 2004.

An Alternative View of design (and planning)

by Allan Kelly

Traditional software development techniques highlight the importance of planning our software through the creation of designs. We often measure our work against plans made before coding starts, and many organisations use adherence to plan as a management control mechanism. Yet just about anyone involved in software development knows that time estimates are usually wrong, and program code doesn't always follow designs produced to start with.

Many in the agile process movement openly question why we bother with plans at all. "Do the simplest thing possible" becomes the only design decision we need to make again.

I'd like to propose that planning is useful, but not necessarily for the reasons we often think it is...

Why Plan?

Although the quote is sometimes attributed to others, I believe it was future US president, General Dwight D. Eisenhower who said:

In preparing for battle, I have always found that plans are useless, but planning is indispensable.

The sentiment isn't restricted to the battlefield, I'm sure many software developers have had recourse to this quote on occasions. What lies behind it is fact that we are not blessed with perfect future vision. Most plans contain assumptions about how the future will unfold, many of these assumptions simply extrapolate from the way things have worked in the past – or how we perceive the things to have worked. Many unknowns, and plenty of unknowables, force us to make assumptions.

Even if all our assumptions turn out to be right, we have no guarantee that our plan is complete. How much detail do we need in our plan? Too little detail and you risk missing something important; too much detail and you'll never get beyond planning – sometimes called "paralysis by analysis."

Some assumptions will be conscious and may be explicitly stated, others will be implicit and undocumented. There will be many implicit assumptions in any development effort, these are derived from our existing knowledge of the technology and business and on the whole offer short-cuts to thinking. However, some of our implicit assumptions will cause problems. Planning is one means by which we can flush out these assumptions and challenge our existing mental maps.

That plans assume foresight, and that foresight may be wrong, is fairly obvious. What is less obvious is that plans also assume communication. Even the best plans can fail because they are not communicated clearly, or the receivers don't act on the information as we expect.

Such problems with planning led Arie de Geus to question the role of planning. In the traditional model planning is a tool which attempts to predict the future, the plans are then used to command and control our activities. In contrast de Geus sees planning as a tool for learning:

So the real purpose of effective planning is not to make plans but to change the microcosm, the mental model that these decision makers carry in their heads. (de Geus, 1988)

Like Eisenhower, de Geus is suggesting that we don't make plans so we can follow them, we make plans to map out the terrain – that is, the problem domain we face. But he also goes further in suggesting that by using planning we can accelerate learning. He suggests planning is a game, a game were we can experiment with different rules and safely make mistakes. The important part of planning is not the output, but the process.

De Geus formulated his ideas as part of the planning group at Royal Dutch/Shell. The head of this group, Pierre Wack, used *scenario planning* to explore the future. Perhaps the best book on scenario planning is Peter Schwartz, *The Art of the Long View*. Schwartz is clear about the role of scenario planning:

Planning Gone Wild

As Schwartz noted, managers "prefer the illusion of certainty to understanding of risks and realities". Yet pursuit of this illusion leads to counter productive results and top-heavy, high-ceremony development systems.

While techniques like *Critical Path Analysis* can be useful in identifying dependencies and foreseeing problems, when carried to the extreme it becomes goal-displacement. We find managers obsessing about Microsoft Project plan charts, re-drawing them, changing activities, re-scheduling, adding resource, removing resource, questioning activities and the estimates behind them.

Putting a manager in the corner with MS Project may be a useful way to keep them out of your hair for a while but so is giving them a piece of paper with "P.T.O." on both sides. Eventually the manager steps away from MS Project and comes to asks you why you estimated this activity at a week, or why the estimate said one day and you took five. People quickly forget the meaning of the word "estimate."

Frequently estimates relate to how long it will take an individual to do a task. Overrun because you helped Jane with her work and suddenly all the talk of "teamwork" is forgotten – even if next week Jane repays the favour and you both complete work early.

Project planning can be a way to drive a wedge between people, forcing them to focus on their own tasks rather than the overall goal. Of course, sometimes you don't need to wedge people apart.

Absent architects separate themselves from projects and teams. In some companies, once an individual reaches a certain level it is believed they can wander in for a few weeks, draw up the blueprints and move on, perhaps returning every now and then to check the plans are being followed.

Since those implementing the plans had no hand in drawing them up their learning curve will be steeper and longer, nor will they be particularly motivated to see the plans come to success. Indeed, as part of their learning process they will probably conduct their own planning process of sorts and may well produce a final product that looks nothing like the blueprint, although, the documentation may say it does.

Scenarios are not predictions. It is simply not possible to predict the future with any certainty. [...] Often, managers prefer the illusion of certainty to understanding of risks and realities. If the forecaster fails in his task, how can the manager be blamed? (Schwartz, 1991, p.6)

How do these ideas play out in software development? Before I attempt to answer this question let's just recap on the two key ideas suggested:

- Firstly, while plans may help us to explore the future, even the best plans will not describe the future.
- Secondly, the planning process is actually a learning exercise, and it is this process which we value, not the plans we produce. The learning that occurs during the process is a result of communication, exploration and the surfacing of assumptions. Importantly, this experience is shared by the whole team.

What Planning Do We Do?

Oranges aren't the only fruit, and project schedules aren't the plans we make. Specifications, flow charts, structure diagrams, pseudo code, UML diagrams, interaction diagrams, and a host of other diagrams all constitute plans we make in advance as a way of exploring our problem and solution domains before we start coding.

In fact, even when we start coding we are still planning. Every function which is written with a stub or is flagged "TODO" is part of a plan, the more we code the more the "plan" becomes an implementation.

Planning can be a point of tension between managers and software developers. On the one hand, some managers understand progress to mean lines of code written – Steve McConnell calls this WISCA syndrome – *Why isn't Sam coding anything?* (McConnell, 1993). On the other hand, excessive planning, document writing, project schedules, and fancy architecture diagrams can act like quick drying cement to stop a project from progressing.

Sometimes we do just jump in and code. Occasionally this is because the problem is so simple the solution appears obvious, or more likely, we've seen the problem before and know a solution that works. Other times the problem is so hideous that we don't know where to start, so we try something. In this mode the code is part of the planning process, we are exploring the terrain by experimentation.

The value of prototyping lies in its role as a planning tool. The prototypes are written for different audiences but typically allow people to learn about the solution before committing themselves to a solution. By viewing the prototype, both developers and clients can accelerate their learning about the solution.

"Test first development" is another form of planning. By considering the test cases before we write any code we are again exploring the problem domain. Planning the tests gives us a chance to improve our understanding before we start coding. Almost as a side effect we get a test suite and save ourselves some time later on.

The traditional view of software design is akin to building development, the plans tell us where to build a load-bearing wall. However, with software we don't always know where the load will occur. For example, it is almost impossible to predict where the

Scenario Planning

One of the most extreme versions of "planning as learning" is that of *Scenario Planning.* In creating a scenario the idea is not so much to forecast the future as it is think what challenges and opportunities you, your team or business may face as the world changes.

Scenario planning has its roots in military planning but has been popularised through its use by Shell and authors like Peter Schwartz. In his model we seek out information which may affect the future. Some of this is knowable right now, e.g. the world's population is growing, X babies were born last year, so in 12 years time there are slightly less than X teenagers. Other information is from "weak signals" and comes from talking to technologists, business people, academics, and other thinkers.

Finding people who have insights and ideas, so-called *remarkable people*, may be a challenge but is not impossible. Once found, their ideas should expose some implicit assumptions and help you imagine a different sort of world.

You sift through this information and look for the underlying forces and the events that are important for your scenario. Then you construct a story that explains the facts, highlights the forces and provides insights. Actually, you may want to construct several scenarios, say a best case and a worst case, but each story must be internally consistent.

Once complete you name each of these scenarios. None of the scenarios you have produced forecasts what *will* happen; they only show what *could* happen.

Stuart Brand suggests that scenario planning can be used in designing buildings. By thinking about how a building may develop in the future we may consider what features are important, what is irrelevant and what obstacles we may be creating in a new construction.

Software development could benefit from these ideas too. Software designers aim for flexible products that can absorb change, can be reused and yet are easy to maintain. Each of these attributes comes at a cost, one answer to this rising cost has been XP's YAGNI – "you aren't going to need it" – approach. The problem is, deciding just what you do need and what you don't need is difficult.

Reality is going to be somewhere between these extremes, but how do we know? Scenario planning offers one way of exploring the future of our software and flushing out real requirements.

Likewise, trying to uncover the risks entailed in your project, or where you can expect change requirements to come from, can be analysed through a scenario plan.

Large framework scenarios used for company strategy and government policy can takes months of work to produce, but it is also possible to run smaller project scenarios to examine specific areas of interest. Even here though, you probably want to conduct some research then schedule several days to analyse what you have gathered, agree the forces and write your stories.

While a team is researching and writing scenarios, they are creating a shared understanding and even a shared language about the problem they face. Communication and learning go hand-in-hand.

performance bottlenecks will be in a complex piece of software – the costs of "premature performance optimization" are widely accepted.

Even if building design was an accurate metaphor for software design it is not without flaws itself. Stewart Brand (1994) has criticised architects and lack of flexibility, and has advocated some alternative ideas (see sidebar on scenario planning.)

Planning as Vision Formation

The activity of writing program code requires us to make design decisions with every line we write: Is a `for` loop more appropriate than `while` loop here? A template or a class there?

Of course, we could draw up more detailed plans to help us, but the more detailed our plans the more the plans are the code. (This is one of the failures of mathematical formal methods, the resulting "specification" can be more difficult to maintain than the actual code.) And at the end of the day, we don't deliver plans, we deliver working code, we want to make our design decisions at the most efficient point, sometimes this is high level, sometimes this is low level.

What we require is a framework that allows us to make all our decisions in a coherent manner. If we have some guiding vision for the system there is less need to examine each decision in minute detail.

Traditionally, we would ask a System Architect to draw up a high-level design for a system. This could be refined by "designers" and implemented by software engineers. The engineers are prevented from making mistakes because the plans control what they do.

However, not only does this model assume that the architect and designers get the design right, but it also assumes the model is communicated with complete clarity and understood by everyone involved in a timely fashion.

How often do we see provisional design decisions become fixed elements of the system? By the time we realise part of our design could be better not only is there too much code to change but there is a bunch of developers who need re-educating.

For a system to remain flexible and soft, it is not only necessary to keep the software flexible but the people must be capable of change too. Thus, we return to de Geus's idea that planning is part of the learning process.

(Notice I say the "people must be capable of change", not "change the people". Often the first reaction of new developers on a software project is to claim the existing code is unmaintainable and the whole thing needs replacing.)

In de Geus's world, everyone is part of the planning process. We plan so that we create a mental model of the system which is shared by everyone. To put it another way, by allowing everyone to participate in the design everyone will buy into the architecture and understand how it affects them.

Ric Holt of the University of Waterloo has suggested that software architecture is most usefully thought of as a mental model shared by the development team. It is more important for the team to hold a common understanding of what is being created than it is to create highly detailed descriptions of technology. Holt's conclusion echoes Conway's Law (1968):

When teaching about or designing software architecture we should always remember that the architecture is intimately intertwined with the social structure of the development team. (Holt, 2001)

And so we return to teamwork. For software development to succeed the team needs to work together. What, you may ask, is the role of the architect here?

The role of the architect, indeed any other manager on the project, is changed when we take this view of planning. They no longer sit in a darkened room and emerge with a completed blueprint of how the system should be. Their role becomes one of facilitator.

Architects may still sit in darkened rooms and think grand thoughts, they may still examine strange new technologies, but they no longer emerge with a plan. Instead they emerge to facilitate discussions, their research may play a part in the architecture and vision created by the team but for a team to truly buy into a vision, and to truly understand the architecture, each team member must have a hand in creating the vision.

Emergent Design

While we may like to think that the plans we make at the start of a project actually describe the system we create the reality is usually different. We find a need for objects that were never included in the object model, the algorithms described by flow charts and structure diagrams turn out to be buggy so the code is different, and refactored code quickly diverges from the plans.

As we develop at the code level a design emerges. To a greater or lesser degree this mirrors our pre-coding plans (assuming we made any). But over time the code becomes the best place to look for design. If we want a high level view of what and how a system works we are better abstracting from the working code than examining blueprints devised before the code was written.

Acknowledging that design is an emergent, ongoing process again challenges the traditional role of design and architecture. However, when we re-perceive design as a learning process through which we create a common vision and understanding of the system, and we re-perceive the architect's role as one of facilitator rather than supreme planner then emergent design is a natural result. Because the design which emerges comes from a group of people rather than an individual the design is shared and understood by all.

What About Plans as Documentation?

Of course, plans have another use, they are the place we turn to first when confronted with a new system. Day one on a new job and we all expect to be given the system design, and usually we find it doesn't exist, or, at best, is out of date.

The fact that plans seldom reflect the realised system has long been known, and famously led Dave Parnas and Paul Clements to write about "A rational design process and how to fake it" (Parnas, 2001). They argue that after building our systems, we should go back and create the documentation we would have created if we had perfect foresight.

Although this may seem a novel idea it suffers from a number of problems, not least that it assumes we will be allowed time to write documents once the development has completed.

More dangerous is the fact that we are introducing an element of dishonesty into the process. No matter how well intentioned our motives we are doing something subversive, is it any wonder that managers ask "Shouldn't you have done that before you started?"

Introducing subterfuge into the process is counter-productive as it also undermines trust.

Rather than fake our plans it is far better to be honest and say "We wrote this after the event." If we want documentation for future developers than we should produce that as a specific task based on the working system.

Unfortunately there are two catches here. Firstly, much of what we learn when developing software is tacit knowledge. It may be shared by the team but it is actually incredibly difficult to write down. The fact that we can codify it at all in program code is pretty remarkable – although often we may not realise we're doing it – implicit assumptions again.

We can try and compensate here by writing copious amounts of documentation. However, this brings us to the second catch which observant readers will have spotted already. Remember de Gues's point about speeding up learning? The more documentation you produce the longer it is going to take new people to come up to speed on the system. Less can really mean more, less documentation can result in more time actually learning about the system.

In fact, copious documentation may make things worse still because we come to rely on words and diagrams. Assuming these are accurate (a big assumption) we have now changed the nature of the issue from one of problem solving to one of applying a documented solution.

However, software development is inherently a problem solving activity. If it wasn't we could automate the process. Therefore, although they may help, documentation and plans never contain the solutions; they may actually be false friends.

Final Thought

One final thought, in the de Geus model of planning as learning it is the institution that learns – where we interpret "institution" in the broadest sense. He says:

And here we come to the most important aspect of institutional learning, whether it be achieved through teaching or through play as we have defined it: the institutional learning process is a process of language development. As the implicit knowledge of each learner becomes explicit, his or her mental model becomes a building block of the institutional model. (de Geus, 1988)

The emphasis on language creation is similar to the pattern community. By developing a language, whether through patterns, planning or scenarios, we create high level abstractions that allow us to discuss complex topics.

Other parallels exist with patterns, like patterns this view of planning seeks to turn implicit knowledge into explicit knowledge, both focus on creating building blocks, pattern writers and scenario planners are directed to focus on forces and particular importance is attached to naming both patterns and scenarios.

How different, and how much more exciting, to view planning this way instead of as a GANTT chart.

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

Bibliography

Brand, S. (1994) *How Buildings Learn: What Happens After They're Built*, Penguin.
Conway, M. E. (1968) *How do Committees Invent?*, Datamation.
de Geus, A. P. (1988) "Planning as Learning", *Harvard Business Review*, 66, 70.
Holt, R. 2001 "Software Architecture as a Shared Mental Model", `http://plg.uwaterloo.ca/~holt/papers/sw-arch-mental-model-010823.html`, Position paper to *ASERC Workshop on Software Architecture*
McConnell, S. (1993) *Code Complete*, Microsoft Press, Redmond, WA.
Parnas, D. L., and Clements P.C. (2001) "A Rational Design Process: How and Why to Fake It" In *Software Fundamentals: Collected Papers of David L. Parnas* (Eds: Hoffman, D.M. and Weiss, D.M.) Addison-Wesley.
Schwartz, P. (1991) *The Art of the Long View*, Bantam Doubleday Dell, New York.

Letter to the Editor(s)

More on Singletons

To the Editor,

There are definitely pros and cons of singleton usage, depending on whether they're used properly or abused. I believe both sides bring valid points to the argument. In modern generic programming, I have experienced great benefits from them.

I have worked on several large-scale projects that have employed a unified singleton system approach. On one such project, we currently have 234 singleton instances. These include such things as specific program state and task objects. The dependencies and life-time issue for all 234 instances are automatically handled for us. We can easily add, and change these instances without fear of a system breakage. It's similar to a free high-performance garbage collection at the architectural level.

Prior to working on such large-scale frameworks without such a system, maintenance work has been a nightmare, even for the outstandingly talented developers. The alleviation has done wonders for preventing memory leaks and keeping our project shutting down properly.

For anyone that is interested in this architectural technique, all the necessary code is available. I've posted an article that shows a simple example of how to use a unified singleton system: `http://daudel.org/code/singleton_usage.html` (will post soon)

Jeff Daudel
`jeffdaudel@yahoo.com`

A Standard Individual: A Licensed Engineer

by Chris Hills

Are you an Engineer? Pause and think before you answer.

Recently I saw one of those TV programs about changing houses. The couple were introduced: She was a "nursing assistant" and he was an "engineer". In fact he was a mechanic. Now, can you imagine the outcry had she been described as a Doctor? Several professional bodies and individual doctors would have complained before the program had got to the commercial break.

Today as I am editing this I am waiting for the "service engineer" to swap out the dishwasher. They assure me he is a "fully trained and qualified engineer". Last time he was here he plugged in a laptop to the dishwasher and set it running. He told me he had started City & Guilds Part 1 but had given up. The laptop then told him the control board needed replacing... It seems that any one can be "an engineer".

In the software industry I have seen people who have taken a short programming course and become "software engineers". Now, you try taking your degree in electronics or software and doing an "architectural appreciation course" and calling yourself an Architect... or a six-month first aid course and call yourself a [Medical] Doctor. There are laws against this but you will have to be a barrister to defend yourself in court. This is because it has been deemed dangerous to have unqualified people as Architects, Lawyers, Doctors, Civil (structural) Engineers, Gas Fitters etc. However, there is no virtually no restriction on embedded engineers no matter how safety critical the work.

My central heating fitter has to be CORGI registered before he can fit a cooker, fire or boiler. This involves passing *and regularly re-passing* legally mandated, and expensive, exams to be able to fit these appliances. Although many of these cookers and heating systems are microprocessor controlled there is no requirement for the programmer to have any form of qualification at all.

The options for some sort of registration, certification or licensing for engineers have been looked at and legislation attempted several times over the last century, from statutory and mandatory licensing in various forms to a purely voluntary system. Strangely, for various reasons, in the past it is the Engineering Institutions that have objected to mandatory systems. Some of the major points are:

1886: The Architects and Engineers Bill was defeated. This was lobbied against by the Institute of Civil Engineers, I.Mech.E. and the IEE. It was at this point the Worshipful Company of Plumbers started the register of plumbers, but this was voluntary.

1919/1920: The Institute of Civil Engineers had come round to thinking it was a good idea to have a statutory register of Engineers but again this was vetoed by the other Engineering Institutes.

1926: Another Engineers Bill for Statutory Registration of Engineers was vetoed by the Civils, Mechanicals and IEE. The reason being that the Institutes felt that they should be the judges of standards not the government.

1943: Again the Government was persuaded not to implement a Register of Engineers qualified to work on public contracts.

1980: The Finnison Report lead to the creation of the Engineering Council and protection for titles Chartered Engineer, Incorporated Engineer and Engineering Technician. Unfortunately the term "Engineer" was not included. The Royal Charter protects these titles with Civil Law. Note this was set up by the Government not the institutes.

1993/4: A veritable library of reports and papers turned up at this point: "Engineering into the Millenium" (Eng Council Steering Group), "The Statutory Question" (Porter), "Report of Licensing of Competent Persons" Working Group, "The UK Engineering Profession: The Case for Unification" (Millman), "Engineers and Professional Self Regulation" (Jordan) and others. Note I will dig out URLs for these as I can. They will be added to the version on `www.phaedsys.org`.

Interestingly these preceding cases all seem to be connected in time with major upheavals and wars. 1886 was the middle of the rapid expansion of British interests in Africa, 1919/20 was the end of WW1, 1926 the General Strike and nationalisation, 1943 WW2 and 1980 was the middle of the Thatcher era, free market and high unemployment. In the early 1990's I recall that we had a recession that no one talked about. I am not suggesting a conspiracy! Just looking at the factors.

That brings us up to the present. There have been another flurry of reports mainly in the last two years. These have culminated in the report of May 2003: "Licensing and the UK Engineering Profession" for the Engineering and Technology Board. You can judge for yourself what the major upheaval behind this new interest in licensing is.

I will hazard a guess at the renewed interest. At this point my employer (`www.hitex.co.uk`) would like me to point out these are my personal views and I am not legally qualified! As with last month's item on Corporate Manslaughter, it is sadly not the professional bodies but the insurance companies who are likely to be (indirectly) behind the changes. Product liability. Money talks. Corporate Manslaughter comes into the report "Licensing and the UK Engineering Profession". Engineering, especially software and embedded systems are playing a larger part in our lives. With the pace of modern life there is more scope for causing more "insured casualties". Perhaps I am just a cynic.

The other problem is partly what are you actually trying to license, certify or register? It ranges from the CORGI type system for gas fitters where there is a legal requirement to pass and continually re-pass exams before one can work, through to a voluntary register such as C.Eng via one of the Institutes, the IEE for example (`http://www.iee.org`). C.Eng is a one off assessment with no reassessment. As long as I pay the dues to the IEE and Engineering Council I remain Chartered.

If you think the C.Eng requirements are difficult gas fitters have to re-take the exams every five years for each category of work the undertake: cookers, fires etc. This can cost up to £5000! In other words £1000 a year. This gives them a card that shows the categories of work they can undertake. It is illegal for anyone to work on gas equipment for gain unless they are CORGI registered.

It has been noted recently that the high costs are proving to be creating problems with the legitimate fitters. It is leading to some non-registered working. This is of course illegal which is the get out for the insurance companies... The claim is void if you used illegal fitters. I am getting cynical again! There is an

HSE paper on where the gas competencies are going at http://www.hse.gov.uk/gas/wg3/wg3_co2.pdf or you can look at http://www.corgi-gas.co.uk. You could also ring up my local CORGI registered gas fitter/central heating expert and ask him about CORGI but you had better be prepared for some strong language!! It's not the regulation so much as the cost.

However for the professions the UK is one of the least regulated countries in Europe. As you would expect Germany and Austria lead the way for Engineers along with Italy and Luxembourg. Most of the others fit in between. Strangely, Scandinavia, home of some well known engineering excellence is also low on regulation for Engineering. However, when it comes to Accountants, Architects, Lawyers or Pharmacists they are right up with the rest, again the UK is trailing behind. There is a study that explores the regulation of professional Engineers in Europe at http://www.europa.eu.int/comm/competion/publications/publications.

There has been an attempt to harmonise the engineering professions across Europe with the Eur Ing (European Engineer) Register run by the European Engineering Federation FEANI (http://www.FEANI.org). This is currently voluntary in the UK and is open to Chartered Engineers. Whilst it is currently little used in the UK it will become more important as time goes on especially if you work with European companies. There is a study on where it is expected to go at http://www.upf.es/dcpis/esf/papers/2bcn.doc. I recommend that all Chartered Engineers should ask their professional Institute for the forms and join the FEANI register. The forms are relatively simple and the C.Eng means you are already at the required level. Also it is inexpensive, I think it is £35 for 5 years registration.

Recently in the USA there have been moves to certify and/or license Engineers to practice. In June 1998 Texas established Software Engineering as a recognised and licensed profession. You get the right to call yourself an Engineer and can offer software engineering services for gain. This is for any software for "engineered systems including embedded systems, real-time systems, mechanical devices, electrical devices and power systems". The requirements are a suitable degree, 16 years experience and references from 9 people 5 of which must be Licensed Engineers. Interestingly the following year in 1999 the ACM decided that it was opposed to licensing. This was for a variety of reasons. The report is at http://www.acm.org/serving/se_policy/report.html.

Several other US states and Canadian provinces are following the lead from Texas. Also in Australia there is a similar "Professional Engineer" or PE title. These are at the level of you must be registered before you can practice for gain. IE on license you cant write SW and get paid for it.

Many of the studies have looked at what it is they want to certify or licence. There have been studies on the impact on the costs to the Engineer, the companies using them, administering the schemes and the economy in general. Also how it will affect the industry as with the gas fitters, railway signalling and aircraft maintenance. The ideas range from (as now with C.Eng.) when you get a suitable qualification and a minimum amount of experience you are registered, through being required to do so much training (the Continuing Professional Development scheme), and on to annual re-testing. The IEE and BCS both looked at running CPD schemes and tried them for some years. They modified them and they seemed to disappear.

Whilst the specific requirement will vary from profession to profession many of these studies call for a unified "Professional Engineer" status. This was partly realised by the Engineering Council having the Engineering Institutes as members and overseeing the Chartered Engineer. The proposed merger of the IEE, IMechE and IIE will make this idea even easier to implement a general Professional Engineer status.

That said, various industries are already expanding their licensing. For example The UK Civil Aviation Authority licences pilots, air traffic control and maintenance engineers. Since 2001 (remember 2001 was the start of the current flurry of licensing reports) this has been via the JAR-66 and JAR-145. The JAR or Joint Aviation Requirement is world wide and is expected to widen its remit to encompass more of the smaller light aircraft that are currently not covered. They are regulated by Maximum Takeoff Mass. See http://www.caa.co.uk and http://www.jaa.nl for the licensing regulations. These licences require (for some levels) Chartered Engineer status and the reports are proposing mandatory amounts of CPD or training that must be undertaken each year. Information on this at http://www.sbac.co.uk/files/newsdocs/84/Interim%20Report%20final.pdf.

The Institution of Railway Signal engineers (http://www.irse.org) have operated a licensing system since 1993... That's another of the dates when licensing came to the fore. They have many categories of licensing that require CPD and re-testing at regular intervals.

The US proposals suggest that only 20% of their IT/software people will be able to gain the licensing for the safety critical work. I have heard a similar figure of 20% Chartered and 80% Incorporated Engineers mentioned by a member of the Engineering Council for the UK.

However it finally ends up, licensing for Professional Software and Embedded Engineers is going to happen in the UK. Across the world and in some sectors within the UK these schemes for registering, certifying or licensing professional Engineers are being strengthened and expanded. The UK will become more regulated if not for engineering reasons then for reasons of insurance and liability. Money and commerce are the most powerful forms of energy there are.

In the UK the government wanted to (wants to?) introduce a professional qualification and license. The obvious starting point is Chartered Engineer. In fact this was part of the original thinking behind the Engineering Council and C.Eng. I think that whatever develops it will come from the C.Eng., especially with the tie in to Eur. Ing. There is no reason for any degree qualified embedded engineer not to join the IEE and go for Chartered Engineer status. See if your employer will assist. Tell them it is tax deductible and they may need staff who are Chartered in the not too distant future.

For embedded engineering, both hardware and software engineers would get a C.Eng. via the IEE (http://www.iee.org), pure software engineers could also talk to the BCS. For Chartered Engineer you require a suitable degree and experience. For older applicants, experience and other qualifications are taken into account so, at the moment, a lack of a degree if you have experience is not a bar. Note: For

[concluded at foot of next page]

A More Flexible Container

by Rich Sposato

The Standard Set

Suppose I want to store pointers to some `Employee` records in a set, and then order that set by `Employee` name. That is easy; just make a functor that compares two `Employee` names, and apply it to a set. The snippet below shows how. The example stores bare pointers, but smart pointers are often a better choice for storing in containers.

```
class Employee {
  public:
    const std::string& GetName() const;
    // ... other functions
};

struct CompareEmployees {
  inline bool operator()(const Employee* l,
            const Employee* r) const {
    return (l->GetName() < r->GetName());
  }
};

typedef std::set<Employee*,
                CompareEmployees> EmployeeSet;
EmployeeSet employees;
```

Now, suppose I want to find a certain `Employee` record that matches any given name. Well, to do that, I have to pass in an `Employee` pointer with the properties I am looking for. This is because the set functions that search – `find`, `count`, `lower_bound`, `upper_bound`, `equal_range`, and `erase` – require a reference to the same type as the key of the set. These functions, except `erase`, are commonly called the "Special Set Operations". Using the example above, I have to pass in a pointer to a bogus `Employee` record. The bogus object only exists as a comparison value for finding the real record. Assuming an `Employee` object can be constructed using just a name, then the code looks something like this:

```
Employee* FindEmployee(
                  const std::string& name) {
  Employee bogus(name);
  EmployeeSet::iterator it
                  = employees.find(&bogus);
  return (employees.end() == it)
                  ? 0 : *it;
}
```

But, what if I can't make an `Employee` object so easily? Perhaps no `Employee` constructor accepts just a name. Or maybe constructing any `Employee` object is so expensive that making a temporary on the stack is not worth the effort. Why is it necessary to make the bogus `Employee` record anyway? It would be so much simpler to just pass in the name itself to the `set::find` function and have it return the iterator to the target object.

Template Member Functions

The `std::set`'s search functions typically require a reference to the same type as the set's key. They look like this:

```
template<class Key,class Compare,class Alloc>
class set {
  // ... other parts of set class
  public:
    template<class Key>
      size_type erase(const Key& x);
};
```

Passing any type into these functions is possible only if the functions themselves are templates. These functions are not templated in the STL's associative containers – and would cause code bloat if they were, which I shall discuss later. The next code snippet shows the declarations of these functions as if they are template member functions. Notice that the `Compare_Type` used for each templated function is not among the templates for the class itself. As long as the `Compare_Type` is comparable to the `Key` type, it can be used by the functor. For completeness, listed below are both

[continued from previous page]

the UK and Europe you will need the IEE not the US IEEE. The IEEE, obviously, cannot confer Chartered Status.

As there is more licensing (and more pressure for licensing) within the UK, EU, USA, Canada, Australia and rest of the world and as embedded SW becomes more integral to most (safety related) parts of modern life, the UK will have to follow the rest into some form of licensing. This is inevitable.

Eventually the term "Engineer" might actually become a respected profession in the UK the same as it is in Germany and Texas. On the bright side, last month being a Chartered Engineer got me a reduction on my house insurance! So it is of practical use now.

Well, can you answer the question: Are you an Engineer? More importantly will you still be able to call yourself an Engineer in five years time?

Chris Hills
`chris@phaedsys.org`
`www.phaedsys.org`

Eur Ing Chris Hills CEng MIEE is a Technical Specialist with `www.hitex.co.uk`.

See the QuEST series of papers on SW Engineering

This article was originally published in Embedded Systems Engineering (`http://www.esemagazine.co.uk/`) and is reproduced here with kind permission of the editor. It is also available with the other columns in this series on `www.phaedsys.org` under the SW Engineering button.

the const and non-const versions. I chose the unimaginative name of flex_set for the container class. Other than the name, and templated member functions, it behaves just like std::set.

```
template<class Key,class Compare,class Alloc>
class flex_set {
  // ... other parts of flex_set class

public:
  template<class Compare_Type>
    size_type erase(const Compare_Type& x);

  template<class Compare_Type>
    size_type count(
            const Compare_Type& x) const;

  template<class Compare_Type>
    iterator find(const Compare_Type& x);

  template<class Compare_Type>
    const_iterator find(
            const Compare_Type& x) const;

  template<class Compare_Type
    iterator lower_bound(
                  const Compare_Type& x);

  template<class Compare_Type>
    const_iterator lower_bound(
            const Compare_Type& x) const;

  template<class Compare_Type>
    iterator upper_bound(
                  const Compare_Type& x);

  template<class Compare_Type>
    const_iterator upper_bound(
            const Compare_Type& x) const;

  template<class Compare_Type>
    pair<iterator,iterator> equal_range(
                  const Compare_Type& x);

  template<class Compare_Type>
    pair<const_iterator,const_iterator>
        equal_range(
            const Compare_Type& x) const;
};
```

This looks good so far, but how will these template member functions know how to compare some arbitrary type to the set's key type? The answer to that lies within the comparison functor, or comparator, used to order the set's elements. The comparator is overloaded to compare an Employee record to a const std::string. There are overloads so the name can be on the right or left side for symmetric comparisons. (Indeed, some member functions of flex_set require both.) The result is shown below along with a more efficient FindEmployee function.

```
struct CompareEmployees :
std::binary_function<const Employee*,
                     const Employee*,
                     bool> {
  inline bool operator()(const Employee* l,
                   const Employee* r) const {
    return (l->GetName() < r->GetName());
  }
  inline bool operator()(const Employee* l,
                const std::string& r) const {
    return (l->GetName() < r );
  }
  inline bool operator()(
                   const std::string& l,
                   const Employee* r) const {
    return ( l < r->GetName() );
  }
};

Employee* FindEmployee(
                   const std::string& name) {
  EmployeeSet::iterator
                    it(employees.find(name));
  return (employees.end() == it)
          ? 0 : *it;
}
```

Okay, this is much better. No need to make a bogus Employee object just to find the real Employee. Just pass in the employee name, and the templated flex_set::find function returns the proper iterator.

For every possible type that can search through the container, simply add that type to the comparator and the compiler automatically makes the templated search functions. Using that idea, the example above can be further overloaded to compare a C-style string as shown here. I typically need only one additional type in the functor besides the key type, so here I would have chosen either std::string or const char *, but probably not both. (Admittedly, having a C-style string overload means I do not have to convert a const char * to std::string before doing the search, so one less object to construct.) Now that I can search using a type other than the key type of the set, I often use the key type only when inserting, and so the Employee-to-Employee comparison in the functor is only used for inserting.

```
struct CompareEmployees :
std::binary_function<const Employee*,
                     const Employee*,
                     bool> {
  // rest of functor as shown above.
  inline bool operator()(const Employee* l,
                         const char* r) const {
    return (l->GetName() < r );
  }
  inline bool operator()(const char* l,
                   const Employee* r) const {
    return (l < r->GetName());
  }
};
```

An Alternate Solution

There is another method for searching through associative containers using types other than the key type. A bunch of wrappers, like that shown below, can be stored inside a `std::set`, and allow us to do searches by name.

```
struct EmployeeWrap {
  EmployeeWrap(Employee*);      // not explicit
  EmployeeWrap(const char*);    // not explicit
  EmployeeWrap(const std:string&);
                                // not explicit

  // Either of these is used, but not both.
  Employee* m_Employee;
  const std::string m_name;

  inline const std::string& GetName(void)
                                    const {
    return (0 == m_Employee)
        ? m_name : m_Employee->GetName();
  }
  bool operator<(const EmployeeWrap& l,
                 const EmployeeWrap& r);
};

typedef std::set<EmployeeWrap>
                        EmployeeWrapperSet;

Employee* FindEmployee(
                  const std::string& name) {
  EmployeeWrapperSet::iterator
             it(employeeWrappers.find(name));
  return (employeeWrappers.end() == it)
        ? 0 : (*it).m_Employee;
}
```

This wrapper has several disadvantages. One is that it requires a special wrapper class to hold all the types needed for comparing. To compare `Employee` records with additional types, those types would have to be added to the wrapper, instead of just overloading the functor as needed for a `flex_set`. The example above has 2 elements, and the `std::string` has to be instantiated even if it is never used. The `sizeof(std::string)` varies from 4 bytes to 28 bytes depending upon the implementation. This increases the overall memory consumed by `std::set` even though all instances of `EmployeeWrap` within the set will not use the `std::string`. Another disadvantage is that the code above constructs an unnamed temporary of `EmployeeWrap` to pass into the `std::set::find`. This construction cost is small, and most of the cost can be optimized away. Lastly, `EmployeeWrap::GetName` imposes a small runtime cost to determine which data member has the name – a cost not required by the `flex_set` method. The argument against the wrapper method becomes: "If `flex_set` is available, then why pay for several costs that are not needed?"

Other Associative Containers

Another alternate solution is, "Why not just use map instead of set?" The `Employee` container will be: `std::map<std::string,Employee*> employees`.

There are two answers for that, one is complicated, and the other is more complicated. The complicated answer is that although that provides the desired ordering, it also requires storing a copy of the `Employee` name as a key for the map. (The `EmployeeWrap` example above also stores one `std::string` with an `Employee` pointer, but `std::map` actually uses each `std::string`.) An intuitive reason to avoid this practice is that the key, `Employee` name, is part of the `Employee` object, and it does not makes sense to store the key separately. A more practical reason is that storing the name separately is inefficient. Another practical reason is that someday, an `Employee` name will change, but the copy won't. That copy is the key within a map, and people will be reluctant to change the key in a map, but may not know that a property of the value is the key, and so change name of the `Employee` without updating the container.

The more complicated answer is that `std::map` cannot provide the same flexible search capabilities as those afforded by `flex_set`. To provide that flexibility for the other associative containers, `flex_map`, `flex_multiset`, and `flex_multimap` are needed, which are based upon same idea. Many STL implementations use the same underlying implementation for all four associative containers. By changing the implementation to make any associative container more flexible, the other three also become more flexible with little extra effort.

Using the templated `flex_map::find` member function allows for searches in a map of `Employee`s to `Assignment`s using just the name. This code shows a `flex_map` of `Employee`s to their current `Assignment`s.

```
typedef flex_map<Employee*,Assignment*,
                 CompareEmployees>
                      EmployeeAssignments;
EmployeeAssignments employeeAssignments;

Assignment* GetEmployeeCurrentTask(
            const std::string& employeeName) {
  EmployeeAssignments::iterator it(
      employeeAssignments.find(employeeName));
  return (employeeAssignments.end() == it)
         ? 0 : it->second;
}
```

If `std::map` were used, then something similar to this is required.

```
typedef std::map<Employee*,Assignment*,
                 CompareEmployees>
                      EmployeeAssignments;
EmployeeAssignments employeeAssignments;

Assignment* GetEmployeeCurrentTask(
            const std::string& employeeName) {
  Employee bogus(employeeName);
  EmployeeAssignments::iterator it(
            employeeAssignments.find(&bogus));
  return (employeeAssignments.end() == it)
         ? 0 : it->second;
}
```

Considerations

CompareEmployees inherits from binary_function for use with adapters such as not2, bind1st, and bind2nd. The adapters will only work with the functor's function that has the types specified as templates for binary_function and ignore the overloaded functions. If an adaptable functor is needed that accepts some other type, the simple solution is to make such a functor. The code below shows a functor that derives from binary_function and can be used with std::string. Assuming that the hourlyEmployees container is sorted by name, the find_if call will locate the first Employee with a name less than the given name. One of the costs of the increased container flexibility is that more functors are needed. But, these additional functors are often needed for other purposes and searches on other containers anyway, such as vector.

```
struct CompareEmployeeToName :
std::binary_function<const Employee*,
               const std::string&, bool> {
  inline bool operator()(const Employee* l,
               const std::string& r) const {
    return (l->GetName() < r);
  }
};

typedef std::vector<Employee*> EmployeeVector;

EmployeeVector hourlyEmployees;
// Populate vector ...
EmployeeVector::iterator it(
      find_if(hourlyEmployees.begin(),
              hourlyEmployees.end(),
              bind2nd(CompareEmployeeToName(),
                      name)));
```

Another consideration is code bloat. Many C++ developers complain that templates cause code bloat, and making template functions out of otherwise normal functions allows for bloat. Instead of having just one flex_set::find function, there can now be several. (My own experience is that I rarely use more than one type of a function, so I am not paying for more than one version of flex_set::find anyway.) Fortunately, most search functions in the associative containers are small and inline, so code bloat will be minimal for them. Unfortunately, some functions in the underlying implementation are not as small, but still manageable. Still, the more flexible search abilities are worth a little extra bloat because temporary objects are no longer created. Since the temporary object is no longer needed, the cost of constructing and destroying it goes away, which may actually shrink the overall code size. Whether the flexible searching is worth the little extra bloat is a decision that must be made for each type and container.

The flex_set is not the ideal container for primitive types. A container of type flex_set<long> allows both flex_set::erase(const long& x) and flex_set::erase(const short& x). The compiler created both functions instead of promoting the short to a long and using only one function. This kind of code bloat can easily be avoided by using std::set<long> instead. A corollary to this is that member functions in std::set should not be templated. (Some simple experiments with changing std::set convinced me that making a separate flex_set class was a better solution.) The flex_set is useful for containers that store objects or store pointers to objects, but I prefer std::set for containers that store primitives.

Instead of making another container class, perhaps std::set itself can be extended by adding additional functions that use predicates. The algorithm functions std::find_if and std::count_if are similar to std::find and std::count except that they use a predicate instead of a value. Would a putative std::set::find_if member function that accepts a predicate work? Not really, because the predicate needs to compare an element to a value, and so std::set::find_if will need to receive both the value and the predicate. Which means the compare value passed into std::set::find_if would have to be constructed. This just reintroduces the original problem of constructing a temporary. Nor could adding predicates to the erase, lower_bound, upper_bound, equal_range, and binary_search member functions of std::set provide any greater efficiency than what flex_set already provides for these functions. Using a unary predicate which stores the compare value does not work either, since unary predicates cannot change the order for the compare and key values, and the predicate must be able to use the compare value on both the left and right side.

Could any other member functions of flex_set be templatized in the same way? Only one other function accepts a reference to a const key type as a parameter, and that function is insert. Inserting anything but a key value is meaningless, so the answer is negative. The special set operations, and one of the erase functions, are the only candidates for becoming templates.

Summary

More flexible variations of the four associative containers are possible by changing the search functions into template member functions. Each data type used as a parameter to these functions requires overloading the comparator to compare these data types to the key value of the container's elements. A caveat that goes with the overloading by type is that each type has to be comparable to the element type. The flexible containers have a beneficial side effect of resulting in more efficient code because named temporary variables are no longer necessary. Nothing is gained by changing std::set or the other STL associative containers. Changing member functions into templated functions causes bloat for containers of primitive types, and passing predicates as parameters into std::set functions does not work.

Rich Sposato
rds@richsposato.com

You may use the source code provided at:
http://www.richsposato.com/software.html
as a replacement for the associative containers provided with the GCC compiler.

Acknowledgements

Many thanks to Don Organ, Gerald Chan, Phil Bass, Alan Griffiths, Juan Carlos Aguilar, and Cherryl Smith for their feedback and insightful comments.

Choosing Template Parameters

by Raoul Gough

Choosing the right parameters for a template can make a significant difference to how useful the template is. In this article, I will present a very simple guideline that, where applicable, can improve a template's flexibility. I will also provide an example of how the standard library itself could have applied this guideline but didn't.

The fundamental idea can be seen in the following example:

```
template<typename Element>
struct inflexible {
  typedef Element element_t;
  typedef std::vector<Element> container_t;
  // ...
};

template<typename Container>
struct flexible {
  typedef typename Container::value_type
                                  element_t;
  typedef Container container_t;
  // ...
};
```

These two templates both define two member typedefs `element_t` and `container_t`, presumably for further use internally (not shown). In the first case, although the template can have any element type, it always uses a `std::vector` as container. The second case is more flexible, since it will work with any container, and any element type, provided that the container has a sufficiently vector-like interface. The principle at work can be stated as follows:

"A template will be more flexible if, instead of internally generating a new type from its arguments, it accepts the generated type directly as a parameter."

Unfortunately, there are some costs associated with this approach, which I will point out first before expanding on the benefits. Firstly, the interface may be less convenient. From the example, client code would have to use `flexible<std::vector<int> >` instead of simply `inflexible<int>`. There is an easy solution, which is to provide a convenient interface once the more flexible implementation is available:

```
template <typename Element>
struct convenient :
             flexible<std::vector<Element> > {
};
```

The second cost is that the documentation for the template will be more complicated. In the example, instead of merely specifying the constraints on the element type, the documentation must now also describe what interface the container type must provide, such as an `operator[]` member function, insert functions and so on. Ideally, the requirements would also be broad enough to allow for future changes in the implementation of the template, for instance switching internally from using `operator[]` to random-access iterators.

Lastly, to inter-operate with a wide variety of argument types, the template implementation will need to be more carefully written. In the example, instead of assuming that `size_t` is the correct type for indexing into the container, the implementation should use `typename container_t::size_type`.

A Familiar Example

To see the consequences of writing a more flexible template, let's take a look at `std::map`:

```
template <class Key, class T, ...>
class map {
public:
  typedef Key key_type;
  typedef T mapped_type;
  typedef pair<const Key, T> value_type;
  // ...
};
```

This should be recognizable as a variant of the first template /`inflexible`/, since it generates the `value_type` (using `std::pair`) from template arguments instead of accepting a `value_type` parameter directly. If I have a user-defined type that has **both** `key_type` and `mapped_type` in the same object, I have a problem which will be familiar to some readers from their own experience. For example:

```
struct person {
  person_id_t m_person_id; //Unique identifier
  std::string m_surname;
  std::string m_other_names;
  // ...
};

void foo () {
  typedef std::map<person_id_t, person> map_t;
  map_t my_map;
  person p;
  person_id_t id;

  // must generate a pair object for insertion
  my_map.insert(map_t::value_type
                       (p.m_person_id, p));

  // lookup by ID and modification are
  // convenient
  my_map[id].m_other_names = "Joe";
}
```

The insertion is a little cumbersome, and duplicates the person ID for every entry in the map. How much of a problem this is in practice depends on the nature of the key type in use, but it is usually a good idea to avoid data duplication where possible. Alternatively, one could choose to use `std::set` and have code like this:

```
struct person_id_less {
  bool operator()(person const &p1,
                  person const &p2) {
    return p1.m_person_id < p2.m_person_id;
  }
};
```

```
void foo () {
  typedef std::set<person, person_id_less>
                                    set_t;
  set_t my_set;
  person p;
  person_id_t id;

  // insertion is convenient
  my_set.insert (p);

  // find requires a person object instead
  // of just the ID
  p.m_person_id = id;

  // mutable access requires a const_cast
  const_cast<person &>(
      *my_set.find (p)).m_other_names = "Joe";
}
```

So `std::set` is easier to use as far as insertions are concerned, and does not duplicate any data, but element lookup and modification are made more difficult. In fact, the `std::set` example has a potentially catastrophic problem, since `my_set.find()` will return `my_set.end()` if there is no match, leading to undefined behaviour from the code. None of these problems are insurmountable, but they do reveal some limitations of the two templates' interfaces.

Now consider an alternative template, `map2`, which applies this article's guideline by accepting the complete value type as a template parameter:

```
template <class Value, ...>
class map2 {
public:
  typedef typename Value::first_type
                                key_type;
  typedef typename Value::second_type
                                mapped_type;
  typedef Value value_type;
  // ...
};
```

This template assumes that the supplied type provides the same interface as `std::pair`, which means that the map implementation needs almost no changes at all. Unfortunately, this also means that the supplied type must have public member variables first and second which contain the object's key and mapped value, respectively. So allowing the client to provide different value types, but requiring a matching interface, hasn't actually achieved very much in this case.

Of course, we don't have to stop there. Having made the decision to accept value types other than instances of `std::pair`, it is fairly natural to consider alternative interfaces. For instance, requiring that the value type provide member functions `get_key` and `get_mapped` would solve most of the problems. It would then be relatively easy to extend the `person` class to provide the necessary interface and store `person` objects directly in a `map2`.

Unfortunately, this assumes that the value type knows in advance that it is going to be stored in a map. Furthermore, it is not very convenient for user defined types that could appear in different maps with different keys (e.g. `person::m_surname` would be suitable as an alternative multimap key). A far better solution would be to accept some additional information via a traits class:

```
template <class Traits, ...>
class map3 {
public:
  typedef typename Traits::key_type
                               key_type;
  typedef typename Traits::mapped_type
                               mapped_type;
  typedef typename Traits::value_type
                               value_type;
  // ...
};

struct person_id_traits {
  typedef person_id_t key_type;
  typedef person mapped_type;
  typedef person value_type;

  static key_type const &get_key(
                   value_type const &val) {
    return val.m_person_id;
  }

  static mapped_type &get_mapped(
                    value_type &val) {
    return val;
  }

  static value_type construct(
                    key_type const &key) {
    // Required by map3::operator[]
    return value_type (key);
  }
};

void foo () {
  // person has an unchanged interface
  typedef map3<person_id_traits> map_t;
  map_t my_map;
  person p;
  person_id_t id;

  // insertion is convenient
  my_map.insert (p);

  // lookup by ID and modification are
  // convenient
  my_map[id].m_other_names = "Joe";
}
```

This template provides convenient interfaces for insertion, lookup and modification. It avoids any data duplication and compares well to the `std::map` and `std::set` versions, which each made some of the operations simple but not others.

Before going on, the construct function in the traits class probably requires further explanation. It is necessary because the `map3 operator[]` accepts just a key as parameter and might have to insert a whole new value into the map. The `std::map` template has a similar constraint, since it defines `operator[]` in terms of `insert(make_pair(key, T()))`, requiring that its parameter `T` be default constructible. This is also quite similar to the lookup problem mentioned for `std::set`, which requires a complete value object in order to search the container. The advantage of `map` (or `map3`) is that this problem only arises in `operator[]` and not the alternative `find` member function.

So the traits-based version provides a good solution, because it means that almost any class can be stored in the `map3` container without internal changes. There is some work involved in writing a new traits class every time, but it is easy enough to emulate the original `std::map` interface for the simple cases that it conveniently supports:

```
template<typename Key, typename T>
struct std_map_traits {
  typedef Key key_type;
  typedef T mapped_type;
  typedef std::pair<Key const, T> value_type;

  static key_type const &get_key(
                   value_type const &val) {
    return val.first;
  }

  static mapped_type &get_mapped(
                   value_type &val) {
    return val.second;
  }

  static value_type construct(
                   key_type const &key) {
    // Required by map3::operator[]
    return value_type (key, mapped_type());
  }
};

template<typename Key, typename T, ...>
struct std_map :
          map3<std_map_traits<Key, T>, ...> {
  // constructors...
};

void bar () {
  std_map<int, std::string> my_map;
  my_map[1] = "hello";
}
```

There are plenty of other applications for a more flexible map. For instance, suppose that we want two different indexes for the same collection of person objects, one which uses the (unique) person ID, and another which uses the (non-unique) surname. A sensible way to do this is to use a reference-counted smart pointer, and maintain two sets of pointers that are sorted by the alternative keys. In our case, client code can re-use its existing traits classes by writing a traits pointer-adaptor template. For example (using the boost reference counted pointer available free from `www.boost.org`):

```
// Adaptor to convert a traits class for use
// via a map3 of boost::shared_ptr values

template<typename PlainTraits>
struct ptr_traits {
private:
  typedef typename PlainTraits::value_type
                          plain_value_type;
public:
  typedef typename PlainTraits::key_type
                          key_type;
  typedef typename PlainTraits::mapped_type
                          mapped_type;
  typedef boost::shared_ptr<plain_value_type>
                          value_type;

  static key_type const &get_key(
                   value_type const &val) {
    return PlainTraits::get_key(*val);
  }

  static mapped_type &get_mapped(
                   value_type &val) {
    return PlainTraits::get_mapped (*val);
  }

  static value_type construct(
                   key_type const &key) {
    return value_type(
          new plain_value_type(
               PlainTraits::construct(key)));
  }
};

// Define two pointer-based indexes using
// different keys
map3 <ptr_traits <person_id_traits> > index1;
multimap3 <ptr_traits <person_name_traits> >
                                    index2;
// ...
index1[my_id].m_other_names = "Joe";
```

Note that the fact that the indexes store reference-counted pointers internally is at least partially hidden from the client code. This is not a complete solution, of course, but goes some way towards one (interested readers may also like to investigate the link given under additional reading). An alternative solution using `std::set` would also be possible, but again is not quite as convenient for searching and modifying. Even ignoring the possibility of `find()` returning `end()`, the assignment from above would look more like this:

```
boost::shared_ptr<person> temp(
                         new person (my_id));
(*my_set.find (temp))->m_other_names = "Joe";
```

[concluded at foot of next page]

From Mechanism to Method: Data Abstraction and Heterarchy

by Kevlin Henney

Trees. Everywhere. Ones with green leaves, ones with family members, ones with files and directories, ones with classes, and many more. Trees – most often upside-down with the root at the top and leaves at the bottom – offer a common and useful mechanism for organizing program elements [1]. Strict hierarchies imply nesting and exclusive containment, e.g., single-inheritance hierarchies and organizational structures blighted by antediluvian management thinking. In common use, the term *hierarchy* also includes DAGs (directed acyclic graphs) that, although hierarchical, are not strictly hierarchies. This is the sense in which I will use it in this article because the world we live and work in more accurately reflects the elasticity of this usage, e.g., multiple-inheritance hierarchies and family trees.

But a hierarchy, strict or otherwise, is not the only way of organizing elements in a program [2]:

> *A program which has such a structure in which there is no single "highest level" ... is called a* heterarchy *(as distinguished from a hierarchy).*

The property that distinguishes hierarchies from heterarchies is that the former is acyclic whereas the latter contains cycles.

Of Types and Hierarchies

When we look at a well-factored program, we see that the data concepts have been abstracted according to their use more often than their representation. Primitive data elements have been grouped and reclassified as information with behavior [3]:

> *As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.*

In C++ we have many mechanisms for organizing our types. The two most obvious are classes – which represent an explicitly named concept made available to the compiler – and template type parameters – where the concept of type is implicit according to usage [4].

Types of Subtyping

Some hierarchies represent internal structural concerns: an object hierarchy whose implementation is layered through composition and forwarding, a function hierarchy where a task is decomposed into smaller tasks that are decomposed in turn, and so on. Type hierarchies, by contrast, reflect external usage concerns.

The articulation of type and subtype concepts relates to the four forms of polymorphism [3] – inclusion, parametric, overloading, and coercion – and the five forms of substitutability in C++ [5] – conversion, overloading, derivation, mutability, and genericity.

There are examples of hierarchies where the structural and type concerns are co-aligned. A class hierarchy, which has a tangible structural dimension, can also be a type hierarchy, so that a pointer to a derived class instance may be used where a pointer to a base is declared. This example is the most commonly quoted form of substitutability, but it is far from being the only one.

Because the number of implicit type conversions the compiler is prepared to string together on your behalf is rather low – only one for user-defined conversions – subtype behavior across type hierarchies based on this form of substitutability tends to be in short hops. For instance, the following definitions describe a relationship where a `text` object is expected a `string` object may be provided, and where a `string` object is expected a `char *` may be provided:

```
class string {
public:
  string(const char *);
  ...
};

class text {
public:
  text(const string &);
  ...
};
```

In the `map3` solution from above, these details are effectively encapsulated the `ptr_traits` template.

Conclusions

In the case of `std::map`, a number of changes were necessary to achieve a real gain in flexibility. However, the first step was to allow the client code to choose their own value type. It is then a fairly obvious improvement to access the keys and mapped values via client-provided functions instead of by using member variables. In this case, a traits class is a convenient way to provide the necessary additional information.

More generally, whenever a template internally generates a new type from its arguments, it is limiting its own flexibility. By accepting the generated type as a parameter instead, the template can become flexible not only in terms of the original parameters, but also in the choice of generated type. There are certain trade-offs in terms of ease of documentation and coding, but potential users of the template may discover unexpected benefits from a more flexible template.

Raoul Gough
raoulgough@clara.co.uk

Thanks to Phil Bass for his comments on two earlier drafts of this article.

Additional Reading

The indexed_set library under development by Joaquín Mª López Muñoz. Google for "indexed_set" or see the recent discussion at `http://lists.boost.org/MailArchives/boost/msg54772.php`

A simple implementation of the `map3` template for demonstration purposes can be found at `http://home.clara.net/raoulgough/map/`

The following will compile happily:

```
text message = "This will compile";
```

But the following will be thrown rudely back in your face:

```
void print(const text &);
print("This won't compile");
            // too many conversions required
```

When considering subtyping and genericity, an actual type must support at least the features required for the template type parameter, so the features of the named type used will inevitably be a subtype of the required features. However, subtyping with generics goes further than just this *formal-actual* relationship, and iterator categories provide a good example. Each category defines a type [6], and the relationships between the categories are in terms of subtyping: anything that can be considered a *random access iterator* will also satisfy *bidirectional iterator* requirements, which in turn also satisfy *forward iterator* requirements, which in turn satisfy both *input iterator* and *output iterator* requirements.

Analysis of Variance

When looking at type hierarchies it is worth taking some time out to understand how some of the properties of substitutability come about.

Consider a base class and a derived class that is also a proper subtype of the base (i.e., uses `public` inheritance from the base, overrides `virtual` functions at least at their level of access in the base, and ensures that any member functions that would otherwise be blocked by members declared in the derived class are made accessible through `using` declarations). What would you expect to find available in the derived class's interface as compared to that of the base class? You'd expect to find either the same set of members or more, but not fewer. If you found fewer, this would break substitutability because you could not use a derived where a base was expected. So although you would have a class hierarchy, it would not be a type hierarchy.

This means that the type interface, considered as a set of operations, is *contravariant* with subtyping: as you descend the hierarchy, narrowing the possible object types you can operate on, the set of operations varies in the opposite direction (i.e., becomes wider). You can also see this with generic types: a pointer, which is the model for random access iterators, supports the same operations as an input iterator, plus many more.

Zooming in on the interface, what about the substitutability with respect to individual operations? As you descend the hierarchy narrowing the possible object types, you also narrow the possible behavior resulting from an operation call. This means that possible results also narrow. In C++, if a virtual function's return type is a pointer or reference, then it can be overridden with a more derived return type. Thus the return type can be *covariant* with subtyping: as you descend the hierarchy, the result type also descends. As an example, consider a simple factory scenario, where an interface class has a Factory Method [7] that offers the creation of instances of a product hierarchy:

```
class product {
  ...
};
```

```
class factory {
public:
  virtual product *create() const = 0;
  ...
};
```

Covariance allows us to capture more accurately the constraint that a specific factory implementation returns a specific product, as opposed to any arbitrary product:

```
class concrete_product : public product {
  ...
};

class concrete_factory
        : public factory {
public:
  virtual concrete_product *create() const {
    return new concrete_product;
  }
  ...
};
```

Covariant return types make sense when it is likely that a user will be accessing the feature through the derived type as opposed to the base. This is not often the case, especially for a factory that is intended to abstract concrete details, but it serves to highlight the public advertisement of the constraint. Either way, it is perfectly type-safe because any code written against the `factory` interface:

```
product *example(const factory *creator) {
  return creator->create();
}
```

is still valid if we reconsider it in terms of `concrete_factory`:

```
product *example(const concrete_factory
                                  *creator) {
  return creator->create();
}
```

And, if this is all there is to writing factories for such products, we can generalize the concrete factory code by mixing two forms of polymorphism, inheritance and templates:

```
template<typename concrete_product>
class concrete_factory
        : public factory {
public:
  virtual concrete_product *create() const {
    return new concrete_product;
  }
  ...
};
```

Any transgression of type safety – if `concrete_product` is not descended from `product` – will be picked up at compile time.

Covariance also applies to other results, such as `throw` specs: an overridden `virtual` function must declare the same `throw` spec or a more restrictive one than the function it is overriding. So, a `virtual` function that does not have a `throw` spec can be overridden by one that does, a `virtual` function that declares an empty `throw` spec can only be overridden by one that also has an empty `throw` spec, and a `virtual` function that promises only to `throw` a base exception type can be overridden by one that promises to `throw` the same, a descendant, or nothing.

What about arguments? Does it make sense to have covariant arguments? There is only one circumstance in which it is safe, and C++ does not qualify for it: if a language supports `out` arguments that can be used only for results. C++ `const` reference arguments can be considered `in` arguments and non-`const` reference arguments can be considered `inout` arguments. Any argument that is effectively `inout` must be *invariant* to be type safe. It is feasible for `in` arguments to be contravariant, but this would greatly complicate C++'s already subtle overloading rules, so the rule is that all arguments remain invariant with subtyping. Some languages attempt to support some covariance with `in` arguments – this is the general case in Eiffel and is present only for array passing in Java and C#, where argument signatures themselves are not permitted to be covariant – but these are basically type system hacks that require significant extra support, typically involving runtime checks.

Covariance and contravariance are not just about declared types; they are more generally about behavior. The promise of behavior for an operation at supertype level must remain the same or be strengthened and become narrower with subtyping. For example, is the following a valid implementation of `factory`?

```
class null_factory
          : public factory {
public:
  virtual concrete_product *create() const {
    return 0;
  }
  ...
};
```

It depends on what the expected result of `factory::create` was promised as. If the promised result at the base class level were simply "returns a `delete`-able pointer," then this permits null pointers. Code written (correctly) against such an interface would cater to this assumption:

```
void consume(product &);
void example(const factory *creator) {
  std::auto_ptr<product>
                     ptr(creator->create());
  if(ptr.get())
    consume(*ptr);
}
```

And so `null_factory::create` could be seen to be a valid specialization of `factory::create` in this case because the behavior is covariant, i.e., narrower and more specific than that of the base.

However, if the promised result at the base class level were specified as "returns a non-null `delete`-able pointer," `null_factory` would break code that worked to this spec:

```
void consume(product &);
void example(const factory *creator) {
  std::auto_ptr<product>
                     ptr(creator->create());
  consume(*ptr);
}
```

And so `null_factory` would not be substitutable for `factory`, and therefore not a subtype.

You can also see the covariant promise in action with iterator types: For an input or output iterator, there is no guarantee that `a == b` implies `++a = ++b`, but for a forward iterator – and subtypes – the behavioral promise is strengthened and this guarantee exists. Conversely, requirements placed on callers of functions follow contravariance.

If you are familiar with *design by contract* [8], you may recognize this as the strengthening of postconditions and weakening of preconditions with inheritance. This variance in design by contract is simply a different expression of the substitutability principle and can be derived from it directly.

Of Types and Heterarchies

Although you want to avoid cycles in your compile-time dependency graph or between threads synchronizing on common resources, there are occasions when heterarchies provide a more appropriate structuring scheme than hierarchies. A function heterarchy is expressed through recursion – either simple recursion when a function calls itself or mutual recursion where another called function calls the original caller. A bidirectional relationship between objects can be considered an object heterarchy. At any point in the execution, which object is considered to be the top-level one depends on the action and context. This is often the case with inversions of control flow such as event notification callbacks.

In a hierarchy there is a unique concept of *up* and *down* – imagine yourself anywhere in a hierarchical structure; there is a strong sense of what is above you and what is below, and a strong separation. In a heterarchy there is no such gravity – imagine yourself in a heterarchy, looking "up" or "down" you can see yourself.

Function recursion and cyclic object relationships are examples of structural heterarchies, but what of type heterarchies? The simplest example of type substitutability with a cycle is between `int` and `double`: one can be substituted where the other is expected. Granted, `double` to `int` is lossy, undesirable, and accompanied by a warning on most compilers, but it does indeed form a type heterarchy. Similarly, if a string class supports both a conversion to and from `const char *` (the former being ill advised, but nonetheless common), it too forms a cyclic substitutability relationship.

However, these two examples are cautionary rather than exemplary, and neither of them involves everyone's favorite class relationship, inheritance.

Touch Base

Imagine that you have customized the `new` and `delete` operators for a class:

```
class workpiece {
public:
  static void *operator new(std::size_t);
  static void operator delete(void *,
                              std::size_t);
  ...
private:
  ...
  static allocation heap;
};
```

Assume that `allocation` is a class that actually provides the appropriate allocation intelligence – optimization for speed, instrumentation for debugging, etc. – and can allocate objects of a size fixed on its initialization and deallocate objects that it allocated:

```
class allocation {
public:
  allocation(std::size_t,
             const std::type_info &);
  void *allocate();
  void deallocate(void *);
  ...
};
```

Its correct use would be:

```
allocation workpiece::heap(
   sizeof(workpiece), typeid(workpiece));

void *workpiece::operator new(
                 std::size_t size) {
  return size == sizeof(workpiece) ?
                heap.allocate() :
                ::operator new(size);
}

void workpiece::operator delete(
                void *ptr, std::size_t size) {
  if(size == sizeof(workpiece))
    heap.deallocate(ptr);
  else
    ::operator delete(ptr);
}
```

The size checks are important because `new` and `delete` are, by default, inherited and consequently may be used on a class whose size does not equal that of the base. The code above ensures that `heap` is used only for the intended size and all other allocations and deallocations are rerouted to the global operators.

If you intend to use `allocation` for the same purpose in any other classes, it would be nice to somehow factor out the code above as a mix-in class so that each class that wanted these services would simply inherit from the mix-in. Something like the following:

```
class allocated {
public:
  static void *operator new(std::size_t);
  static void operator delete(
              void *, std::size_t);
  ...
private:
  ...
  static allocation heap;
};

class workpiece : public allocated {
  ...
};
```

Except not. All classes derived from `allocated` will share the same `static heap`:

```
class command : public allocated {
  ...
};
```

A `command` may not have the same size as a `workpiece` and will certainly not have the same type, so not only is `heap` shared but it also becomes impossible to initialize or use correctly, as the places marked `???` in the following code indicate:

```
allocation allocated::heap(
   sizeof(???), typeid(???));

void *allocated::operator new(
                 std::size_t size) {
  return size == sizeof(???) ?
                heap.allocate() :
                ::operator new(size);
}

void allocated::operator delete(
                void *ptr,std::size_t size) {
  if(size == sizeof(???))
    heap.deallocate(ptr);
  else
    ::operator delete(ptr);
}
```

Even attempting to factor out the constant `sizeof` expression will not solve the problem. It serves only to highlight it more sharply.

Essentially the problem is that the mix-in cannot be made fully independent of any derived class: there is a lingering dependency on the name and size of the derived class. This creates a cycle that can be broken by treating the downward dependency as a parameter of variation, and therefore something to template [9]:

```
template<typename derived>
class allocated {
public:
  static void *operator new(std::size_t);
  static void operator delete(
              void *, std::size_t);
  ...
private:
  ...
  static allocation heap;
};
```

Given this, the implementation code can now get at the name and size of the type:

```
template<typename derived>
allocation allocated<derived>::heap(
    sizeof(derived), typeid(derived));

template<typename derived>
void *allocated<derived>::operator new(
                          std::size_t size) {
  return size == sizeof(derived) ?
                heap.allocate() :
                ::operator new(size);
}

template<typename derived>
void allocated<derived>::operator delete(
                void *ptr, std::size_t size) {
  if(size == sizeof(derived))
    heap.deallocate(ptr);
  else
    ::operator delete(ptr);
}
```

The last piece falls into place with the mixing-in itself:

```
class workpiece : public allocated<workpiece>
{
  ...
};

class command : public allocated<command> {
  ...
};
```

Now the derived class inherits from a class that knows about it, but without any hardwired coupling that prevents its use as a general solution. Each parameterization of `allocated` results in a distinct type with its own code and data, which is exactly what was required.

This is the increasingly well-known *Self-Parameterized Base Class* or *Curiously Recurring Template* pattern – the latter being an evocative description of its recognition [10] and the former being a more appropriate contemporary name based on its structure.

In Good Shape

The `allocated` class template demonstrates an example of heterarchical structure involving class relationships. However, it is not a type heterarchy because no useful properties of type, except for `typeid` and `sizeof`, are used: it steadfastly avoids dealing with objects.

Consider a simple shape hierarchy, with the usual suspects `ellipse` and `rectangle`. It makes sense to be able to copy such objects polymorphically by cloning them:

```
class shape {
public:
  virtual shape *clone() const = 0;
  ...
};
```

```
class ellipse : public shape {
public:
  explicit ellipse(const ellipse &);
  virtual shape *clone() const {
    return new ellipse(*this);
  }
  ...
};

class rectangle : public shape {
public:
  explicit rectangle(const rectangle &);
  virtual shape *clone() const {
    return new rectangle(*this);
  }
  ...
};
```

There are a couple of observations to make on this:

- Cloning is effectively a reflexive *Factory Method*, where the product and factory are of the same class.
- The `explicit` copy constructor means that although `ellipse` and `rectangle` support explicit copying – as seen in `clone` – the general case of passing and returning them by copy is not supported, i.e. the identity conversion is disabled. `shape` objects are heap-bound and live in a class hierarchy, rather than being value-based objects for which casual copying makes sense.
- The implementations of `ellipse::clone` and `rectangle::clone` are suspiciously similar, differing only in the class to which they refer.

It would be nice to factor out the common source structure, except that it is the structure as opposed to the verbatim code that needs factoring out. Again, there is a cyclic type dependency, and this time more entrenched because of object creation and copy construction. And again, a *Self-Parameterized Base Class* offers a way to break the cycle [11]:

```
class shape {
public:
  virtual shape *clone() const = 0;
  ...
};

template<typename derived, typename base>
class cloner : public virtual base {
public:
  virtual base *clone() const {
    return new derived(
         static_cast<const derived &>(*this));
  }
  ...
};

class ellipse : public cloner<ellipse,
                              shape> {
  ...
};
```

[concluded at foot of next page]

CheckedInt: A Policy-Based Range-Checked Integer

by Hubert Matthews

Recently, I wanted a short example to show the canonical form for operators on value classes. In other words, I wanted to show how post-increment should be related to pre-increment, how `operator+=` and `operator+` fit together, which functions should be members and which not, and so on. Having also been reading Alexandrescu's excellent book *Modern C++ Design*, I decided to make this exercise a little more interesting (for me and for the students) by incorporating something about policies and generic programming. What came out was a small range-checked integer type called `CheckedInt`. Although nothing remarkable, it turns out to be both flexible and useful, and something that in retrospect I could have used myself on several occasions.

For those who are not so familiar with operators, this class shows how all of the arithmetic operators can (or maybe even should) be implemented in terms of one fundamental operation: `operator+=`. This ensures consistency between operators, thereby avoiding potential surprising arithmetic inconsistencies. (For reasons of space, I show only the addition-based operations. Implementation of the others is left, in time-honoured fashion, to you, Gentle Reader™.)

[continued on next page]

```
class rectangle : public cloner<rectangle,
                                shape> {
  ...
};
```

There are a few observations to make on this solution:

- To reiterate a property of heterarchies: you really can see yourself if you look down.
- An attempt to parameterize `cloner` with a non-derived class will cause a compile-time error because the derived class will not be substitutable for the base in the code.
- If you consider cloning to be a reflexive version of *Factory Method*, this solution mirrors the templated `concrete_factory` in a reflexive way: in each case, the product is the first template parameter.
- Another adaptive template technique, *Parameterized Inheritance*, is used to define the appropriate base class.
- Inheritance uses a `virtual` base class to accommodate other applications of this reflexive mix-in style, but without introducing repeated inheritance issues.

Conclusion

A template technique that is becoming increasingly common has been taken and explored within the conceptual framework of substitutability. From its early sightings [12] and subsequent exploration [10] to the present day [13, 14], the *Self-Parameterized Base Class* pattern has found increased applicability in expressing cyclic type relationship problems once they have been recognized as such.

There are many techniques that combine two main forms of generalization – templates and inheritance – and a number of them have been used in this article. However, a *Self-Parameterized Base Class* has the distinction that it unifies the forms under the heading of substitutability but not of hierarchy.

Kevlin Henney
kevlin@curbralan.com

Notes and References

[1] Michael Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* (Addison-Wesley, 1995).
[2] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid* (Penguin, 1979).
[3] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, December 1985.
[4] Kevlin Henney. "From Mechanism to Method: Good Qualifications," *C/C++ Users Journal C++ Experts Forum*, January 2001, http://www.cuj.com/experts/1901/henney.htm
[5] Kevlin Henney. "From Mechanism to Method: Substitutability" *C++ Report*, May 2000, also available from http://www.curbralan.com
[6] Note that these generic requirements-based types are sometimes referred to as concepts. However, use of this term is inadvisable because of its impressive accuracy without any precision whatsoever: everything in software development can be considered a concept, but only a few things can be considered types. Generic requirements define – quite precisely – types.
[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).
[8] Bertrand Meyer. *Object-Oriented Software Construction, 2nd edition* (Prentice Hall, 1997).
[9] James O. Coplien. *Multi-Paradigm Design for C++* (Addison-Wesley, 1999).
[10] James O. Coplien, "Curiously Recurring Template Patterns," *C++ Report*, February 1995.
[11] Kevlin Henney. "Clone Alone," *Overload 33*, August 1999.
[12] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples* (Addison-Wesley, 1994).
[13] Kevlin Henney. Email correspondence with Angelika Langer, July 2000, http://www.langer.camelot.de/IOStreams/forum.htm
[14] Klaus Kreft and Angelika Langer. "Effective C++ Standard Library: Curiously Recurring Manipulators," *C/C++ Users Journal C++ Experts Forum*, June 2001, http://www.cuj.com/experts/1906/langer.htm

This article was originally published on the C/C++ Users Journal C++ Experts Forum in August 2001 at
http://www.cuj.com/experts/documents/s=7992/cujcexp1908Henney/
Thanks to Kevlin for allowing us to reprint it.

For those already familiar with operators, the policy aspect is more interesting. What should happen when you try to take a range-checked integer or enum out of its defined range or even just modify it? For our range-checked integer, a number of possibilities sprang to mind:

- allow silent overflow
- throw an exception
- saturate at the limit value
- saturate at the limit and log the event
- wrap around using modular arithmetic
- log the event for debugging purposes
- etc.

This little class template allows us to choose which behaviour we want by means of a policy class. Allowing silent overflow is the default for integers so there's no need to write a class for that. Throwing an exception when straying from the promised range is possibly indicative of a programming error. Saturating at the limit could be useful for a digital volume control; one that sticks tenaciously to 10 when you try to set it to 11. And wrapping around is very useful when dealing with ring buffers, dates, etc.

This is a simple example of feature-driven modelling and domain analysis, as described in *Generative Programming and Multi-Paradigm Design for C++* where families of types are created with variations described in policies.

So, here's the abbreviated code:

```
template <int low, int high>
class OutOfBoundsThrower {
public:
  static int RangeCheck(int newVal) {
    if(newVal < low || newVal > high)
      throw std::out_of_range(
                          "RangeCheck failed");
    return newVal;
  }
};

template <int low, int high>
class ModularArithmetic {
public:
  static int RangeCheck(int newVal) {
    while(newVal > high)
      newVal -= high - low;
    while(newVal < low)
      newVal += high - low;
    return newVal;
  }
};

template <int low, int high>
class SaturatedArithmetic {
public:
  static int RangeCheck(int newVal) {
    if(newVal > high)
      newVal = high;
    else if(newVal < low)
      newVal = low;
    return newVal;
  }
};
```

```
template <int low, int high,
     template <int, int>
     class ValueChecker = OutOfBoundsThrower>
class CheckedInt :
           protected ValueChecker<low, high> {
  int value;

public:
  explicit CheckedInt(int i = low) :
    value(RangeCheck(i)) {}

  CheckedInt& operator+=(int incr) {
    value = RangeCheck(value + incr);
    return *this;
  }

  CheckedInt& operator++() {
    *this += 1;
    return *this;
  }

  const CheckedInt operator++(int) {
    CheckedInt temp(*this);
    ++*this;
    return temp;
  }

  CheckedInt& operator-=(int incr) {
    *this += - incr;
    return *this;
  }

  operator int() const {
    return value;
  }

  CheckedInt& operator=(int i) {
    value = RangeCheck(i);
    return *this;
  }

  const CheckedInt operator+(
             const CheckedInt& other) const {
    return CheckedInt(*this) += other;
  }
};
```

Construction and Member Functions

Note that the constructor is, like most single argument constructors, marked as `explicit`. This is to avoid implicit conversions that muddy the type system. Consider what would happen with `CheckedInt<0,10>(5) + 27` if 27 could be explicitly converted. What should its template parameters be? Should it throw an exception? An explicit constructor avoids these problems and forces us to state what we want to happen. The explicit nature of object creation is particularly useful when we wish to constrain the underlying `int` to a given range as we do not want to create erroneous values. Some might bemoan the inability to write `CheckedInt<0,10> ci = 5;` but I think that safety is more important than ease this time. Choosing `low` as the default parameter is purely arbitrary and it is arguable that we should force the user to give an initial value anyway.

When going in the opposite direction, i.e. from a `CheckedInt` to an `int`, there is no danger of breaking any constraints so we can safely use a user-defined conversion – `operator int()` – so that `CheckedInt` appears in a read-only context to behave like an `int`. This allows us to use all of the existing infrastructure for `int`s such as `operator<<`, `operator==`, `operator<`, etc. We can now do things like `CheckedInt<0,10>(5) + 27` with impunity and no fear of exceptions.

One small fly swims in the ointment of `operator+=`. There is the possibility that the expression `value + incr` might overflow causing undefined behaviour. This would cause an unexpected problem with saturated arithmetic if someone tried to add a very large number to an instance that was already at its upper limit. Alternative implementations, such as templating the underlying arithmetic type, are possible but more complex.

The more astute of you might have noticed that `operator+` is unusual: it is a member and it is `const`. The normal advice is to make `operator+` a non-member to allow for implicit conversion of the left-hand operand. However, since we have specifically disallowed that conversion there is no reason not to make it a member and save ourselves a lot of typing! We also return a `const` value to prevent modification of a temporary whilst still allowing it to be bound to a reference.

Templates Versus Object-Oriented Interfaces

An interesting difference in style arises with generic programming rather than a more traditional object-oriented approach. With O-O, one usually ends up with an interface that is the union of all of the sub-interfaces, whereas with a templated version the interface is usually minimal and the intersection of features. This is primarily because with an O-O interface you can combine only those things that you design *a priori* to be combinable, i.e. they must implement all of the stated interface, which can lead to a lot of clutter and "just in case" methods. With templates, you can combine anything that works *a posteriori*. Thus, templates provide compile-time signature-based polymorphism in a manner more reminiscent of Smalltalk than the "one size fits all" of Java interfaces or C++ abstract base classes.

Inheritance Versus Delegation

Here I have inherited from the policy class rather than delegating to it. Altering the class to use delegation instead:

```
template <int low, int high,
   class VC = OutOfBoundsThrower<low,high> >
class CheckedInt {
public:
   explicit CheckedInt(int i = low) :
                  value(VC::RangeCheck(i)) {}
```

moves us towards a traits-style approach, which some might consider to be cleaner. It is also more digestible by older compilers. In this case because the policy has no state of its own – it is just a wrapper for a function – there is little to choose between the two approaches. The `ValueChecker` in effect is a compile-time functor analogous to a combination of `bind2nd()`, `logical_or()`, `less<int>()` and `greater<int>()`.

Legacy Compilers and Binding-Time Issues

Those of us who have to tiptoe around non-standard or ancient compilers will know that template template parameters are off limits. So, how can we adapt `CheckedInt` to be usable? One way is to pass `low` and `high` to the `RangeCheck` function at run-time. This has the nice effect of making `ValueChecker` a non-templated class and thereby eliminating some of the compilation problems. Another would be to have a static member of the class that held a pointer to a free function to do the range check. This implementation would also allow the policy to be changed at run-time, turning the class into a classic run-time version of Strategy pattern rather than a compile-time version.

What we are doing is making binding-time choices. By delaying binding from compile time to run time we trade efficiency for the ability to use simpler constructs. We can even change the parameters, so that we could alter the valid range of an object. Whether we wish to do this depends on requirements. As more programmers begin to understand the parallels between different C++ mechanisms and as compilers get better, I believe that we will see binding time become a major design topic, leading people into both feature-driven modelling and domain analysis.

Extensions and Additions

Possible extensions include making the underlying type a template parameter, as well as extending the `RangeCheck` function to take the original value as a run-time parameter as well. This would allow us to implement propagating NaN (not a number) behaviour, where if the new value is outside the range we set the value to an out-of-bounds value and keep it there. This is a little bit like the effect of floating-point NaNs which propagate "NaNness" into the results of any calculation.

Summary

I hope this little class template is both useful and instructive. It raises a number of common design issues – relationships between operators, implicit v. explicit conversions – and some others – binding times, policies, implicit v. explicit interfaces, etc – that are less widespread but which I believe will become increasingly common with time. If anyone uses `CheckedInt`, particularly with policies other than these, I would be most interested to hear your experiences.

Hubert Matthews
hubert@oxyware.com

Acknowledgements

My thanks go to Kevlin Henney and Andrei Alexandrescu for comments on this article.

Bibiliography

Alexandrescu, Andrei, *Modern C++ Design*, 2001, Addison-Wesley

Czarnecki, K & Eisenecker, UW, *Generative Programming*, 2000, Addison-Wesley

Coplien, JO, *Multi-Paradigm Design for C++*, 1999, Addison-Wesley